Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

FRIDAY, OCTOBER 7, 2011

There's a New Scripting Tool in the Box

The introduction ExtendScript support in FrameMaker 10 offers new possibilities for augmenting the built-in capabilities of the Adobe FrameMaker product. FrameMaker has long been extensible but prior to release 10, extending FrameMaker meant mastering the intricacies of the Frame Developer Kit (FDK).

The FDK extends the capabilities of FrameMaker by allowing for the creation of C language plug-ins. Such plug-ins have great power as they can be used to mimic just about any action a user might take, thus offering vast automation potential. FDK plug-in also can be integrated into the FrameMaker menu structure making FDK commands operate as if they were part of the out-of-the box product.

Developing such plug-ins requires knowledge of the C programming language. The development process entails writing, compiling, and debugging code using Microsoft Visual C development tools. The FDK developer needs to master a voluminous set of objects, properties, functions along with complex data structures. As C has no built-in garbage collection, FDK programmers also have to deal with memory management, a task which when bungled is all but sure to lead to unpredictable and catastrophic crashes.

The challenges of FDK plug-in development meant that only a small fraction of Frame users attempted plug-in creation. Plug-ins became something large corporations commissioned and less well-heeled users purchased from third-party developers.

The addition of ExtendScript scripting to FrameMaker 10 not only makes extending FrameMaker potentially easier, but it also moves FrameMaker further along the road to full-fledged integration with the larger set of Adobe products. (Adobe acquired FrameMaker in 1995.) A major step came in release 9 with a major UI redesign that made FrameMaker look and feel more like Adobe products such as Photoshop and InDesign. I cannot help but believe that this bodes well for FrameMaker's future and for its ability to "play well" other Adobe products.

My focus here, however, is not on interoperability but on the promise implicit in ExtendScript support that FrameMaker users now will be able to construct scripts that have the power of FDK plug-ins without the need for heavy-duty programming skills.

I would be remiss not to mention that the third-party created FrameScript has long offered an alternative to FDK plug-in development. (See http://www.framescript.com for more information.) I will *not*, however, be discussing FrameScript. My focus is strictly on taking a very close look at Adobe's ExtendScript implementation of FDK functionality.

I have been an FDK programmer for many years. I started learning the FDK, even before its public release, as I prepared the first FrameMaker training class for Frame Technology. Having taught FDK developer classes too many times to count, in many states and several countries, and having developed numerous FrameMaker plug-ins for corporate users, the `F_Api` prefix that starts of FDK functions is hard-wired in my brain. Thus, I cannot help but view the ExtendScript toolkit from the perspective of an FDK programmer.

But while, I may frequently reference FDK code, my intention is that anyone who is new to the ExtendScript FrameMaker toolkit will find this blog helpful. I plan to look at common-place tasks in FrameMaker customization and show how they can be scripted using ExtendScript. I will provide code examples and commentary. When I digress into areas of interest only to FDK programmers, I will provide ample warning. I will assume that you know end-user FrameMaker and that you have some scripting experience but not that you are familiar with ExtendScript or even JavaScript. I will be posting irregularly, as I come up with useful examples and information. So, please stay tuned.

Posted by Debra Herman at 9:55 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, OCTOBER 8, 2011

Easier to Code? Easy to Code?

The Adobe FrameMaker 10 *Scripting Guide* states that FrameMaker ExtendScript scripts "act as wrappers to the FDK and hide the complexity of using FDK functions from users." From this we can also infer that these scripts are no more powerful than the plug-ins one might create with the FDK. Thus, for an FDK programmer to migrate to this new scripting tool, there should be (after some learning curve) a greater ease of tool development. What might account for this ease?

The scripts are created in ExtendScript, essentially an augmented version of JavaScript. That means they are interpreted rather than compiled. There is no need to mess with include files or to link libraries. That should be a boon to those just getting started although they offer no benefit to experienced FDK programmers.

The FDK plugin development requires that you have the Microsoft Visual C++ development environment. To create scripts, you are likely to want to have Adobe's ExtendScript Toolkit. The ExtendScript Toolkit is easier to get started with but does it provide sufficient support for debugging? For now, I leave this unanswered. (I need to do some more heavy duty development work before I make a judgment.)

What about the code? Here is Adobe's "Hello World!" written in ExtendScript:

```
Alert("Hello World!", 1); /* FM Alert With OK and Cancel buttons */
```

Here is the same program written for the FDK:

```
VoidT F_ApiInitialize(IntT init)
{

     switch(init) {
     case FA_Init_First:
     F_ApiAlert((StringT) "Hello World!", FF_ALERT_OK_DEFAULT);
     break;

}
```

At first glance, ExtendScript wins the ease of use battle hands down. (All that extra FDK code is telling FrameMaker that the alert should appear before FrameMaker shows itself to the user. The alert box produced has **OK** and **Cancel** buttons with **OK** as the default choice.)

Will this advantage hold up in the face of more complex programming tasks? That is the $64,000 question. In future posts, I will be taking common FDK programming paradigms and rewriting them in ExtendScript. With a few of those under my belt, I will have a more informed opinion.

Posted by Debra Herman at 8:36 PM

No comments:

Post a Comment

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, OCTOBER 9, 2011

Document Architecture

To create ExtendScript scripts that manipulate the content of FrameMaker documents, you need to know the parts that make up a FrameMaker document and how they inter-relate.

If you are familiar with FrameMaker from an end-user perspective, you already know something about how the FDK, and ExtendScript access the content within FrameMaker documents. To write a script, however, you have to stop thinking about how an end user might accomplish a task and think instead about how a program might access an object that it wants to inspect or change.

Here are some things you need to know before getting started with scripting:

- FrameMaker edits documents. Groups of documents can be aggregated into books.
- Documents can be structured or unstructured. (Structure in FrameMaker is really just an overlay on an unstructured FrameMaker document. More about that later.)
- The name FrameMaker is a reference to the idea of a frame as a container for document content. All content is a FrameMaker document is ultimately part of some frame.
- Frames can contain flows. Flows can be thought of as the containers for rivers of text. Such text can span page boundaries.
- Frames can be anchored or unanchored. Anchored frames are tied to a position in text. Unanchored frames are placed in a specific coordinate position on a page.
- Frames contain graphic objects and other frames. These can be the circles, rectangles, and other shapes you might draw with the built-in graphics tools. Graphics also can be anchored and unanchored text frames (containers for sequences of paragraphs that can themselves contain text), text lines (such as the callout text you might use in an image), insets (imported graphics), groups of graphic objects (created using the Group tool), equations created with the Equation Editor.

Why do you need to know what objects exists and how they inter-relate? Very simply, to work with these objects using the ExtendScript Tool Kit (ESTK), you need to locate them first. Once you have "hold" of an object, you can learn about its characteristics (properties) and change them as desired.That is the key to the usefulness of the ESTK.

Posted by Debra Herman at 2:24 PM

No comments:

Post a Comment

Comment as:

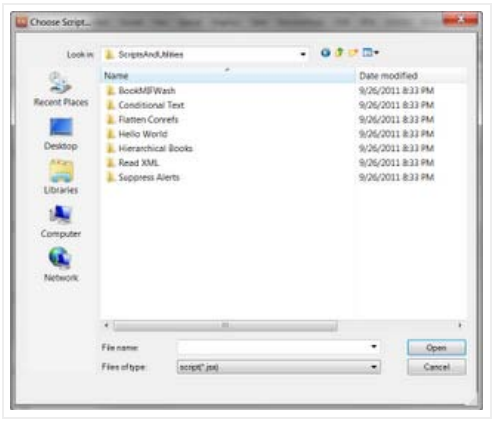
Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

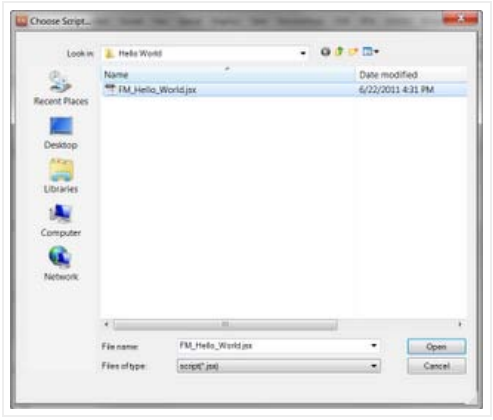
SUNDAY, OCTOBER 9, 2011

Running a (Very Simple) Script

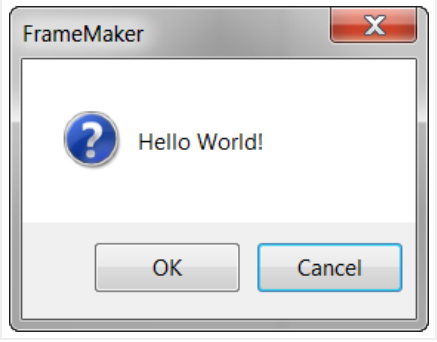
FrameMaker 10 ships with a number of sample ESTK scripts. You can run these scripts using the `File>Script>Run` command. After selecting that command, you are prompted to locate the script of interest. The samples can be found in your FM install directory under `Samples/ScriptsAndUtilities`.



Double click on the `Hello World` folder to display the `.jsx` file that is the actual script.



Click on `Open` to view the dialog box opened by the script. This is the result of the trivial script, `Alert("Hello World!", 1);`



Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, OCTOBER 10, 2011

Hello Again

Simple though it may be, the `Hello World` code has one cryptic aspect: why the second parameter with the value "1"? The comment tells part of the tale. But it omits the fact that, in this case, `Cancel` is the default option.

```
Alert("Hello World!", 1); /* FM Alert With OK and Cancel buttons */
```

If I change the parameter to "0", I get an alert box with `OK` and `Cancel` buttons with `OK` as the default.

Consulting the documentation (page 649-650 of the Scripting Guide) reveals that the first parameter is the message to display and the second determines the type of message box uses. A less cryptic version of Adobe's `Hello World` is shown here using the appropriate constant value:

```
Alert("Hello World!", Constants.FF_ALERT_CANCEL_DEFAULT);
```

My own Hello World might better be written:

```
Alert("Hello World!", Constants.FF_ALERT_OK_DEFAULT);
```

There are actually six different alert box types, each with a corresponding constant. The number is easier to type but the constant, once learned, will make your code clearer and more maintainable.

By the way, the `Alert()` method returns `0`, if the user clicks OK and `-1` otherwise.

Posted by Debra Herman at 3:58 PM

No comments:

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, OCTOBER 11, 2011

Getting Ready to Use FrameMaker with the ESTK

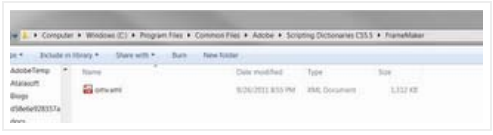
You could, if you are truly brave, author your `.jsx` script files in any text editor. You could then run them using the Scripts menu Run command. But the rest of us will want to use the ExtendScript Toolkit. That means downloading the ExtendScript Toolkit OMV. (OMV is short for Object Model Viewer).

This means downloading and installing the OMV file. You can navigate to the download using the built-in Help. Look for the FrameMaker Developer Center. For convenience, I am including a link here:

FrameMaker 10 ExtendScript Toolkit OMV

The setup instructions tell you where to locate the OMV file. In my default FrameMaker 10 installation, I found that I had to create the directories named. When installing the OMV, be sure to place it in the location that matches the version of the ESTK you intend to use.

Here is my OMV file:



Posted by Debra Herman at 3:24 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, OCTOBER 11, 2011

The Active Document

One way or another, nearly all FDK plug-ins and ESTK scripts, work with documents. (The most likely exception to this rule might be a script that worked strictly with book level information.)

Before a script can do anything with a document two things need to be true:

- The document must be open. (Open and visible on screen are *not* the same thing.)
- The script need to grab hold of the document's object or, in FDK parlance, identifier.

Once you have found the object that represents a document, you can access the objects that make up that document (the paragraphs, graphics, flows, frames and so forth).

It is possible to open documents using a script but for now, lets work with a document that is already open. In fact, lets work with not just any open document but with the document that has the user focus. This is the document where words typed at the keyboard would go and to which any other user action would apply. That document is known as the *active document*.

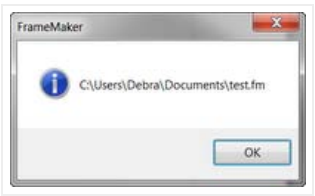
Use the following code to get the document object:

```
var doc = app.ActiveDoc;
```

NOTE: app is the application object. Its function is essentially the same as that of the `FV_SessionId` identifier in an FDK plug-in.

Use the following code to determine the document's file name and then display that name using an alert:

```
var name = doc.Name;
Alert(name, Constants.FF_ALERT_CONTINUE_NOTE);
```



Remember, your active document must be saved to have a name!

Posted by Debra Herman at 4:30 PM

No comments:

Post a Comment

Extending FrameMaker

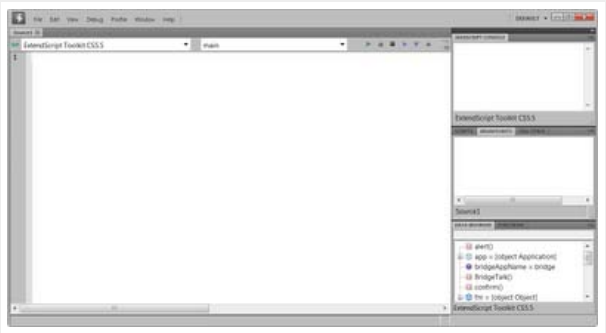
Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, OCTOBER 12, 2011

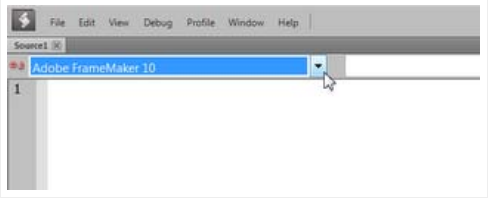
Hooking up the Toolkit

To use the ExtendScript Toolkit with FrameMaker, do the following:

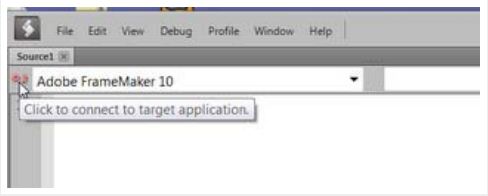
1. Launch ExtendScript. The following window appears.



2. In the popup at the upper-right choose Adobe FrameMaker 10.



3. Click on the red icon (two links, one broken) to connect the toolkit to FrameMaker 10.



4. If FrameMaker is already running, ExtendScript connects to the running instance. If not, the following dialog appears. Click `Yes` to connect.

5. To verify that your installation works, enter the code for `Hello World` in the code window. Click on the run icon (sideways green triangle) to launch the script.

6. Locate the FrameMaker application to view the alert. (Yes, it is annoying that it does not automatically come to the top.)

search

Publish Preview

Posted by Debra Herman at 3:45 PM

1 comment:

Ian Proudfoot November 3, 2011 4:55 PM

FrameMaker should come to the top if you set up the ESTK as follows:
1. Open the preferences
2. Select the debugging page
3. Check "Bring target application to front"

It works for me about 90% of the time...
Ian

Reply

Add comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, OCTOBER 12, 2011

Paragraphs, Ordered and Unordered

Frame documents contain paragraphs, graphics, flows, frames, and other stuff. With the exception of elements (as in XMLtype elements in structured documents), these things are organized into lists.

Some of these lists are ordered and some have no predictable order. Consider paragraphs. Every FrameMaker document has a list of paragraphs that is unordered. At first glance, you may be surprised that this list has no order but a bit of reflection reveals that paragraphs have order within flows and not within documents. Thus a script can get the paragraphs within a specific flow in the order in which they appear in the document or it can get the paragraphs in the document (a larger set) in an unpredictable order.

What can you do with paragraphs that are out of order? You can do just about anything that can be done to a paragraph that does not depend upon its specific location within the document.

If you were to look at the Frame Developer Kit (FDK) documentation, you would see that there is no table that separates the ordered and unordered lists. How can you tell which is which? The simplest method is to check to see if the list is linked backward and forward (that is, if there are *next* and *prev* properties). Such a list is ordered. A list with only a *next* property is unordered.

Lets get specific using paragraphs as an example. (To look this up yourself, see the section for `FO_Pgf` in the *FDK Programmer's Reference*.) In the FDK the paragraph object has these "object pointer" properties:

```
FP_NextPgfInDoc  //Next paragraph in the unordered list for this doc
FP_NextPgfInFlow //Next paragraph in the ordered list for this flow
FP_PrevPgfInFlow //Previous paragraph in unordered list for this flow
```

So can you access the start of these lists?

- In the case of the unordered list, the first is a document property, `FP_FirstPgfInDoc`.
- In the case of the ordered list, the first is the *text frame* property `FP_FirstPgf`. There is also a `FP_LastPgf` property allowing for traversal of the list from last to first as well as first to last. (If you were expecting a flow property, you are likely not alone. Paragraphs within flows will get a more detailed treatment in a later entry.)

All of these properties have an analog in the scripting world. If you rely on rule of thumb given in the FramemMaker *Scripting Guide*, you can easily calculate the appropriate name.

> Remove the `FP_` prefix before using the properties in scripts.

```
FirstPgfInDoc //First paragraph in unordered list for this doc
NextPgfInDoc  //Next paragraph in the unordered list for this doc

FirstPgf      //First paragraph in ordered list for this flow
LastPgf       //Last paragraph in ordered list for this flow
NextPgfInFlow //Next paragraph in the ordered list for this flow
PrevPgfInFlow //Previous paragraph in unordered list for this flow
```

Too many years of FDK programming led me to revert back to FDK docs and remembered knowledge. If you are completely new to all of this, you can skip FDK-think and go directly to the *Scripting Guide*. Chapter 4 contains the *Object Reference*. Here you will also learn the important fact that the data type for these properties is Pgf. (In the FDK, it would be `F_ObjHandleT`.)

Posted by Debra Herman at 5:24 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

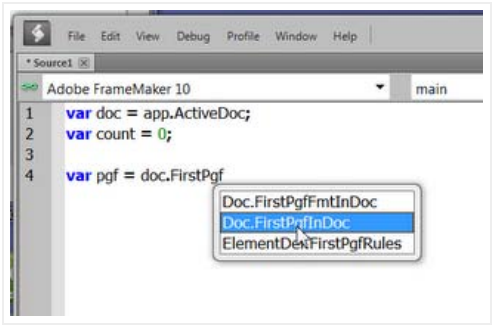
THURSDAY, OCTOBER 13, 2011

Counting Paragraphs in a Document

You may have considering counting words but who wants to counts paragraphs? Perhaps no one. But to count paragraphs, you have to find them all. Therefore counting paragraphs provides a simple exercise in traversing a list. Once the paragraphs are counted/found, something more interesting can be learned *about* them or done *to* them.

Note: You can access a word counting utility from the File>Utilities>Document Reports command. This utility is implemented as an FDK plug-in.

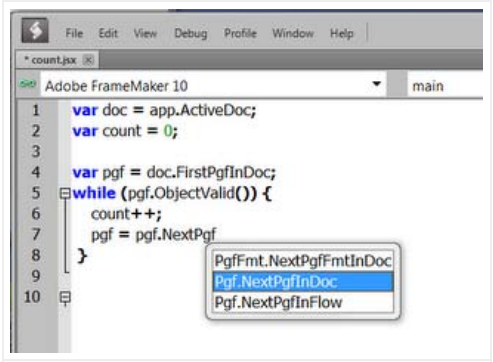
Start by getting the active document and by setting the `count` variable to `0`. Next get the first paragraph in the list of unordered paragraph. It is a document property. If you start typing from memory, the ESTK nicely prompts you with the possible completions. (NICE!)



The scripts calls for a `while` loop. Stop when Frame runs out of paragraphs. Here is where the FDK programmer in me gets into trouble. I am tempted to write `while (pgf)` assuming that `pgf` goes to zero when there are no more. That does not hold in ExtendScript. (If you goof like I did you will end up with FrameMaker in an infinite loop. At that point kill it using the Windows Task Manager.)

`while (pgf.ObjectIsValid())` determines whether the object exists in the current document.

Each time through the loop the count is incremented, the attempts to find a next paragraph. Once again, the auto-completion feature in the ESTK is helpful.



Now all that remains is to report the count.

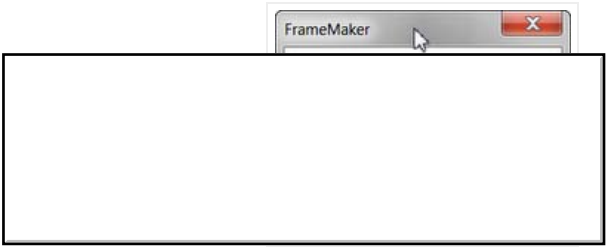
```
count = count+''; //concatendate empty string to convert
```

```
Alert(count);
```

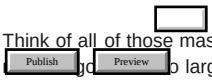
The completed script is shown here:

```
var doc = app.ActiveDoc;
var count = 0;

var pgf = doc.FirstPgfInDoc;
while (pgf.ObjectValid()) {
    count++;
    pgf = pgf.NextPgfInDoc;
 }

count = count+''; //Concatenate empty string to convert
Alert(count);
```

If you run this script with the `Portrait` template, you get the following:



Think of all of those master and reference page paragraphs and you will understand how the [illegible] large.



Posted by Debra Herman at 7:54 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, OCTOBER 15, 2011

Counting Paragraphs in the Main Flow

While there may be times when you need to work with all of the paragraphs in a document, it is much more likely that you want to get the paragraphs in a few in the order in reading order. You could, if you wish, get all of the paragraphs in all of the flows in order. This post deals with the most likely scenario: getting all of the paragraphs in the main flow in the order in which they would print.

What is the main flow? The main flow is just like any other named flow in a document. It is the default flow for the language version use. Thus:

- When a table of contents or index is generated, the generated text is put into the main flow of the the generated document.
- When the `Compare Documents` command is run, the summary text is placed in the main flow.

For English documents, the default flow is `Flow A`. (If there are several autoconnect flows in the document with the default flow tag, the main flow is the one in the *backmost text frame on a Right master page*.)

The FDK provides a document property that allows plug-in to get the main flow easily: `FP_MainFlowInDoc`. In ExtendScript, it is thus `MainFlowInDoc`.

The `FirstPgf` and `LastPgf` properties allow scripts to locate the first and last paragraphs in a flow but, oddly enough, these are *not* flow properties but text frame properties. So to find the start (or end) of this list, you need to get the first text frame within the main flow (or the last text frame if you wish to navigate from bottom to top.) Use the `FirstTextFrameInFlow` flow property (or `LastTextFrameInFlow` flow property).

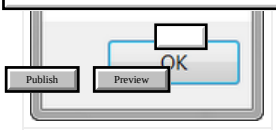
`Note:` Text frames do not span pages. When a text frame is full, FrameMaker creates a new text frame on the next page. That frame is connected to and remains within the same flow as the original text frame.

To get successive paragraphs, you need not worry about text frame boundaries (assuming they are of no interest to you). You can simply get the next paragraph (or previous) in the flow you are traversing.

```
var doc = app.ActiveDoc;
var count = 0;

var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;
while (pgf.ObjectValid()) {
    count++;
    pgf = pgf.NextPgfInFlow;
    }

count = count+''; //Concatenate empty string to convert
Alert(count, Constants.FF_ALERT_CONTINUE_NOTE);
```

Now if I run the script on the empty **Portrait** template I get the following:

search



The end of flow marker counts as a paragraph. The paragraphs on the master and reference pages as well as in headers and footers, are no longer counted.

Try it yourself on a longer document with multiple paragraphs of text in the main flow.

Posted by Debra Herman at 2:25 PM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, OCTOBER 16, 2011

Getting and Setting a Paragraph Property

It is an easy change to have the count paragraphs script update the paragraph instead. In this example, all paragraph text is underlined. All you need to know to make this change is the name of the `Underlining` paragraph property and its possible values:

- `Constants.FV_CB_NO_UNDERLINE`
- `Constants.FV_CB_SINGLE_UNDERLINE`
- `Constants.FV_CB_DOUBLE_UNDERLINE`
- `Constants.FV_CB_NUMERIC_UNDERLINE`

Here is the updated code:

```
var doc = app.ActiveDoc;

var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;
while (pgf.ObjectValid()) {
    pgf.Underlining = Constants.FV_CB_SINGLE_UNDERLINE;
    pgf = pgf.NextPgfInFlow;
    }
```

Note: The change just made was to the *paragraph* and not the *paragraph format*. There is an asterisk next to each format name in the status area at the lower left as shown below. This reflects the fact that the format has been over-ridden.

While you are unlikely to want to add underlining to a paragraph, perhaps you might want to remove it. To do so, first determine if the paragraph is underlined and then change the property setting as desired.

```
var doc = app.ActiveDoc;

var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;
while (pgf.ObjectValid()) {
    if ( pgf.Underlining = Constants.FV_CB_SINGLE_UNDERLINE)
        pgf.Underlining = Constants.FV_CB_NO_UNDERLINE;
    pgf = pgf.NextPgfInFlow;
    }
```

Note: This script removes any underlining not applied with a character format. Underlining that stems from the application of a character format remains as is.

Posted by Debra Herman at 1:32 PM

No comments:

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, OCTOBER 17, 2011

Working with Paragraph Formats

A paragraph's format can updated by setting its `Name` property to a new value.That action changes the name of the paragraph format but it does not add a new paragraph format to the catalog. (If the new format already exists, that catalog format would be used.)

It is possible to programmatically add new formats to the catalog using a script. Paragraph formats are "named" objects. In the FDK, `F_ApiNewNamedObject()` can be used to create one. In ExtendScript, you need to use `NewNamedObject()` and pass in the object type and the name for the new object.

A paragraph format created in this way has a default set of properties. You can set the properties to the desired state one at a time. In a later post, I will discuss other ways to give an object a desired set of properties.

In summary, this example creates a new paragraph format named `Para` and then changes any paragraph in the main flow with the format `Body` to the format `Para`.

```
var doc = app.ActiveDoc;
var tframe = doc.MainFlowInDoc.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

//create a new paragraph format
var pgfFmt = doc.NewNamedObject(Constants.FO_PgfFmt, "Para");

while (pgf.ObjectValid()) {
    if ( pgf.Name =="Body")
        pgf.Name = "Para";
    pgf = pgf.NextPgfInFlow;
    }
```

Posted by Debra Herman at 6:53 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, OCTOBER 18, 2011

Updating a Paragraph Format

This post looks at how to update an existing paragraph format and then, having made several updates, apply those updates to all paragraphs that use that format.

As the paragraph format exists, the script uses its name to get its object:

```
var pgfFmt = doc.GetNamedObject(Constants.FO_PgfFmt, "Body");
```

The script then changes several properties of the format:

- The paragraph is set to be auto-numbered.
- An autonumbering string is defined. (This is the definition string.)
- Capitalization is set to upper case.
- Change bars are turned on.

These changes only update the paragraph format. To apply the changes to all of the existing paragraph formats, it helps to think about how an end user would accomplish this task. (File>Import Formats ... from the current document into itself with check Paragraph Formats and remove Other Format/Layout Overrides chosen.)

In ExtendScript, this translates to the SimpleImportFormats method with the following parameters:

```
var err = doc.SimpleImportFormats(doc, Constants.FF_UFF_PGF|
Constants.FF_UFF_REMOVE_EXCEPTION);
```

IMPORTANT: The FrameMaker ESTK documentation calls for ORing the constants. The FDK documentation calls for a *bitwise or* and, based on my observations, that appears to be correct here as well.

Use the Portrait template in your testing or any other template that has a Body paragraph format defined. Your file *must be saved* or the import of formats cannot succeed.

```
var doc = app.ActiveDoc;
var tframe = doc.MainFlowInDoc.FirstTextFrameInFlow;

var pgfFmt = doc.GetNamedObject(Constants.FO_PgfFmt, "Body");
if (pgfFmt.ObjectValid) {
    pgfFmt.PgfIsAutoNum = true;
    pgfFmt.AutoNumString= "<n+>. ";
    pgfFmt.Capitalization = Constants.FV_CAPITAL_CASE_UPPER;
    pgfFmt.ChangeBar = true;

    var formatFlags = Constants.FF_UFF_PGF |
Constants.FF_UFF_REMOVE_EXCEPTIONS;
    doc.SimpleImportFormats (doc, formatFlags);
}
else
    Err("Format not found");
```

Posted by Debra Herman at 4:28 PM

No comments:

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, OCTOBER 19, 2011

Where's the text?

Having delved into paragraphs and paragraph formats, text might seem like the next logical topic. I am going to defer the discussion of text for a while for a number of reasons.

First, in FrameMaker documents, text can appear in paragraphs or in text lines. Text lines are actually a type of graphic object. So graphics need to come before text.

Text also contains anchored objects such as markers, tables, anchored frames. A basic understanding of these is helpful when working with text.

Finally, in the FDK at least, text is managed quite differently from everything else. The `F_ApiGetText()` function works with text items rather than text objects. These text items contains the actually text characters but also information on changes in text formatting. This makes working with text a bit harder than working with other document objects. I suspect the same will hold true with scripting.

So, if you are hoping to hear about text in detail, hang onto your hats, it is coming!

Posted by Debra Herman at 11:50 AM

No comments:

Post a Comment

Comment as:

search

neMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, OCTOBER 19, 2011

Counting Insets

Insets are just imported images. They can be still images or video. From the point of view of the FDK or ESTK scripting, they are a subtype of graphic object.

The list of all graphics in a document is an unordered list. The first graphic in the list is found using the document property `FirstGraphicInDoc`. Subsequent graphics can be found using the graphic object property `NextGraphicInDoc`.

`Note`: A single insert can have multiple facets.

`Note`: The only graphics you can work with in flow order are those that are anchored in text. It is possible to work with selected graphics, graphics on a page by page basis, or those within a particular frame.

Rectangles and other shapes that can be created with the FrameMaker built-in drawing tool are also graphic objects as are text frames both anchored and unanchored. Thus, graphic objects have a `type` property that makes it possible to distinguish between the differing types.

This script defines a function to count the graphics in the document of interest,

```
var doc= app.ActiveDoc;
var count = CountGraphics (doc);
Alert(count, Constants.FF_ALERT_CONTINUE_NOTE);

function CountGraphics(doc)
{
    var count = 0;
    var graphic = doc.FirstGraphicInDoc;
    while (graphic) {
         if (graphic.type == Constants.FO_Inset)
            count++;
         graphic = graphic.NextGraphicInDoc;
    }
    count = count+''; //concatenate empty string to convert
    return (count);
 }
```

Posted by Debra Herman at 7:11 PM

No comments:

Post a Comment

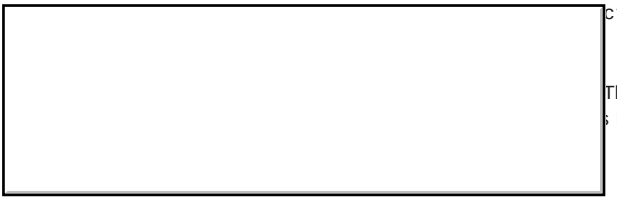
Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, OCTOBER 20, 2011

A Peculiar Fact about Frame Graphic Object Properties

If you poke around in the ESTK documentation, you will find that objects such as `FO_Inset` and
`ctor`, and others that



This fact is clearer in the
s have these properties but

Publish y Preview man at 2:14 PM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

FRIDAY, OCTOBER 21, 2011

Import by Reference

Two important pieces of information about any imported graphic are :

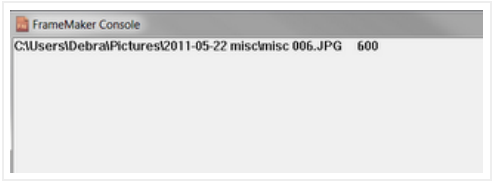
- The full pathname of the imported file.
- The DPI at which the file was imported.

The following code uses `Err()` to display this information in the FrameMaker console.

The file name is just a string while the DPI is an integer. To display it in the console, requires that it be converted to a string.

```
var doc= app.ActiveDoc;
ListGraphics (doc);

function ListGraphics(doc)
{
     var graphic = doc.FirstGraphicInDoc;
     while (graphic) {
         if (graphic.type == Constants.FO_Inset) {
             Err(graphic.InsetFile);
             Err("     ");
             Err(graphic.InsetDpi + "");
             Err("\n");
             }
         graphic = graphic.NextGraphicInDoc;
      }
 }
```

Here is the output from a file with just one imported graphic:



Adding the code line `graphic.InsetDpi = 2400;` within the loop, resets the image's DPI making it smaller. The console now shows:

NOTE: The console does *not* get cleared between successive runs of your script. Closing it clears the content.

Posted by Debra Herman at 11:52 AM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, OCTOBER 25, 2011

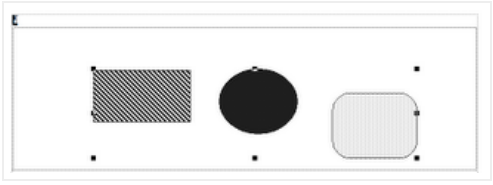
Grouping Graphics

The following example shows how to group graphic objects within an anchored frame. It works on the selected graphic. If two or more graphics are selected, it asks the user to select just one.

Recall that grouped graphics within FrameMaker can be manipulated as a unit. While this script works with anchored frames, it could have worked with unanchored frames or both.

For example, assume that the selected graphic contains the following shapes, each ungrouped as indicated by the fact that they can be selected individually.

The script groups them into a graphic that can be worked with as a unit as indicated by the change in the graphic handles.



The script first determines if there is in fact a selected graphic using the document property `FirstSelectedGraphicInDoc`. If there is indeed such a graphic, it then determines if that graphic is an anchored frame.

Once a selected anchored frame is detected, the script creates a group object using the document method `NewGraphicObject()`. It passes in the object type and the ID of an anchored frame.

The script them loops through the list of graphics in the frame. Each object found is assigned the group parent just created. When the script completes the graphics within that frame are part of a single group.

```
doc = app.ActiveDoc;
aFrame = doc.FirstSelectedGraphicInDoc;

if (!aFrame.ObjectValid()) {
    Alert("Select an anchored frame and try again.",
Constants.FF_ALERT_CONTINUE_WARN);
    }
else if (aFrame.type != Constants.FO_AFrame) {
    Alert("Selected object is not an anchored frame. Adjust your
selection and try again.",
    Constants.FF_ALERT_CONTINUE_WARN);
    }
else {  // we have an anchored frame
    group = doc.NewGraphicObject(Constants.FO_Group, aFrame);
    graphic = aFrame.FirstGraphicInFrame;
    while (graphic.ObjectValid()) {
        graphic.GroupParent = group;
        graphic = graphic.NextGraphicInFrame
        }
     Alert("All graphics in this anchored frame have been grouped.",
```

```
        Constants.FF_ALERT_CONTINUE_NOTE);
    }
```

NOTE: If there are already grouped graphics within the anchored frame, they are treated no differently than the rectangle, ellipse or oval. The group becomes a nested group.

NOTE: Although they are graphics, anchored frames cannot be grouped. If there was a nested anchored frame, setting its group parent would have no effect.

Posted by Debra Herman at 1:24 PM

No comments:

Post a Comment

Comment as:

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, OCTOBER 25, 2011

Ungrouping graphics

Ungrouping a graphic is just a matter of setting its `GroupParent` property to zero.

```
graphic.GroupParent = 0;
```

The graphic being ungrouped may itself be group. Ungrouping its members is easy, however, as when looping through all of the graphics in a frame, you find both ordinary graphics and groups.

This code ungroups any grouped graphics within an anchored frame by setting all `GroupParent` properties to zero.

```
group = doc.NewGraphicObject(Constants.FO_Group, aFrame);
graphic = aFrame.FirstGraphicInFrame;
while (graphic.ObjectValid()) {
        graphic.GroupParent = 0;
        graphic = graphic.NextGraphicInFrame
        }
```

Posted by Debra Herman at 2:17 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, OCTOBER 27, 2011

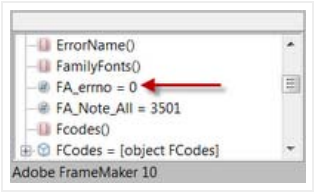
Debugging with Error Codes

The FDK and the ESTK use the *global* variable `FA_errno` to indicate whether or not a plug-in or script has executed correctly in the sense that all function calls were well-formed and executed correctly in a formal sense.

Initially `FA_errno` has the value `0` (Constants.`FE_Success`). `FE_Success` has the value `0`. All other possible values of `FA_errno` are negative integers.

The value of `FA_errno` remains `FE_Success` so long as no error occurs.

You can view the current value of `FA_errno` in the ESTK Data Browser. Just be sure to connect to FrameMaker first.



If an error takes place, `FA_errno` is reset to a non-zero value. That value remains until another error causes it to be reset or until you reset it yourself.

The ESTK lists 111 error code constants on pages 59-65 of the FrameMaker 10 *Scripting Guide*. Individual functions that set error codes have detailed information within the function documentation.

What sort of things cause an error code to be set? For example, you might

- Attempt to set a value for a read-only property. (`Constants.FE_ReadOnly`)
- Attempt to get the value of a property that the object in question does not possess. (`Constants.FE_BadPropType`)
- Pass an invalid object identifier to a function. (`Constants.FE_BadObjId`)
- Pass the wrong object type of a function (`Constants.FE_NotFrame`, is one possibility)
- Request an operation that cannot be carried out.(`Constants.FE_BadOperation`)

Posted by Debra Herman at 3:48 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

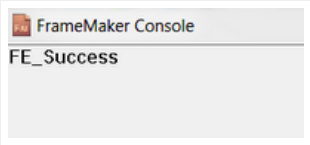
FRIDAY, OCTOBER 28, 2011

Error codes: when are they actually set?

The function `PrintFAErrno()` displays the current value of `FA_errno` not as a number but as a constant. This is convenient in understanding what went wrong in your code.

I did a little experimenting and found some surprising results. If I give the following correct code, I get an error code of `FE_Success` as expected.

```
var doc = app.ActiveDoc.FirstPgfInDoc;
PrintFAErrno();
```



If I change my code as shown below, I expect a non-zero error code as documents do not have the property `NextPgfInDoc`. (That is a paragraph property.) My tests show, however, that `FA_errno` remains `FE_Success`.

```
var doc = app.ActiveDoc.NextPgfInDoc;
PrintFAErrno();
```

So what is going on? I am not quite sure. While I distinctly recall seeing actual error codes as I developed other examples, I find that right now I am unable to get anything other than `FE_Success`. My error? ESTK problem? I fear I have made some subtle mistake. I will be keeping this issue in mind as I go forward and will report on my findings.

Posted by Debra Herman at 9:05 PM

2 comments:

frameexpert November 3, 2011 4:46 PM

Hi Debra, What you actually have is a syntax error in your script, not a method error. The error codes get triggered when a method fails, but not for syntax errors. Did the ExtendScript Toolkit complain about the improper syntax?
--Rick

Reply

Debra Herman November 3, 2011 5:05 PM

Hi Rick,

I do not see a syntax error. My thinking is very influenced by what I am used to seeing in the FDK. I was excepting BadPropNum as an indication that object in question does not have the property named.

Reply

Add comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, OCTOBER 29, 2011

Working with Text

Sooner or later, you are likely to write a script that works with text. Some things you might want to do are:

- Determine the content of a range of text.
- Add text to a document.
- Remove text from a document.
- Alter the formatting of text.
- Work with objects that are anchored in text such as tables, markers, or graphic.

Where is text found?
Lots of places including:

- table cells
- structure elements
- flows
- footnotes
- paragraphs
- text frames
- text lines
- sub columns
- text insets of various types
- variables
- cross references

What is found in text?
Text contains the alphabetic, numeric, and special characters that make up document content. But text can also contain anchored objects. Those are the tables, anchored frames, markers, cross references that can be inserted at locations between characters.

Text is different
Text, unlike documents, paragraphs, graphics and just about everything else that makes up a FrameMaker document is not an object. There is no `FO_Text` object. Instead, text within paragraphs, text lines and other objects that contain text are processed as text items. (The details will be the subject of a later post.)

Text is formatted
Text all appears in a given font, at a particular size and style. It contains line breaks and page breaks and other information that reflects how it looks on the page. When you work with text, you will need to be able to determine how it is formatted and, if desired, change that formatting. Text items also give you the information needed to tackle those tasks.

Locations and Ranges
To work with text, you need to understand the concepts of text location and text range. You can think of a text location as analogous to an insertion point and a text range as analogous to a text selection. (But a text range can be selected or not selected!)

Posted by Debra Herman at 4:00 AM

No comments:

Post a Comment

search

neMaker

Publish Preview

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, OCTOBER 29, 2011

Calculating Offsets

To work with text you need to understand how FrameMaker uses offset. While your scripts can work with text in flows or even documents, offset is always from either the start of a paragraph of text line.

Counting offset
Use the following rules in calculating the offset of a location in text:
• The start of a paragraph or text line has offset 0.
• Each character adds offset of 1.
• Anchors of any type have an offset of 1.
• Element boundaries have offset of 1.

The following have no offset:
• Paragraph begins.
• Line begins and ends.

But, *paragraph ends have offset* and must be selected to change paragraph defaults. The end of flow marker cannot be selected either by an end user or by a script. It has no offset.

Posted by Debra Herman at 1:17 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, OCTOBER 29, 2011

Adding Text at a Location in Text

Just as user typing appears at the insertion point, a script adds text at a text location.

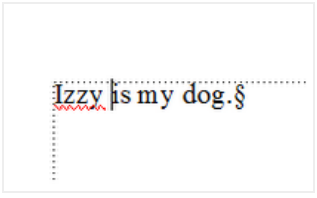
A text location consists of an object identifier (either an paragraph or a text line) and an offset from the start of that object.

The following example inserts text at the beginning of the first paragraph in the active document.

```
var doc = app.ActiveDoc;

var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

var tLoc= new TextLoc(); //create the text location object
tLoc.obj = pgf; //make it a paragraph
tLoc.offset = 0; // insert at the start of the paragraph
doc.AddText (tLoc, "Izzy ");
```

If you start out with the following document

and run the script, you get



If you change the text location to 3 (`tLoc.offset = 3;`), you get the following:

Posted by Debra Herman at 7:52 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, OCTOBER 30, 2011

Deleting a Text Range

Deleting text means working with a text range. The range specifies the starting and ending paragraph or text line and the offset from each.

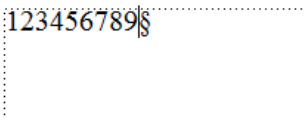
While a user must select text in order to delete it, a script does not need to do so.

NOTE: A text range cannot span text lines but can span paragraphs. Any selection within a flow must be contiguous but can begin in one paragraph and end in another.

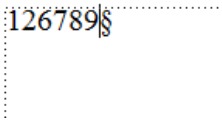
The following example shows how to delete the third through the fifth characters in the first paragraph of the main flow. Selecting the third character means starting the range at an offset of 2, just before that character.

```
var doc = app.ActiveDoc;
var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

var tRange= new TextRange();
tRange.beg.obj = pgf;
tRange.beg.offset = 2;
tRange.end.obj = pgf;
tRange.end.offset = 5;
doc.DeleteText(tRange);
```

Here are the before and afters of a simple test:



Before running the script



After running the script

Posted by Debra Herman at 8:39 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

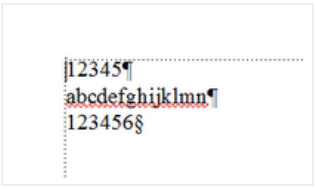
TUESDAY, NOVEMBER 1, 2011

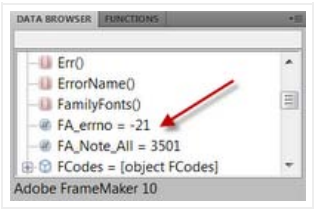
Error Code When Deleting a Range and Some Buggy Behavior

If you attempt to delete a range that is beyond what exists in the document, you get an error code as is appropriate.

```
var doc = app.ActiveDoc;
var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

var tRange= new TextRange();
tRange.beg.obj = pgf;
tRange.beg.offset = 2;
tRange.end.obj = pgf;
tRange.end.offset = 100; // END OFFSET OUT OF BOUNDS
doc.DeleteText(tRange);
```

As the ending offset is out of range and I get the error code `-21` (`FE_OffsetNotFound`) for the document shown. No text gets deleted.





If I change the code to `tRange.end.offset = 10;`, the following happens:

In summary, the "345" and the end of paragraph marker get deleted from the first paragraph and the "abcd" from the line that follows. That is eight characters in all as line ends and start paragraphs do not have offset (at least not in the FDK). The error code is now back to 0 (`FE_Success`).

I have two problems with this behavior:

- The delete should not have gone beyond the paragraph specified.
- The offset calculation is incorrect.

Posted by Debra Herman at 12:38 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, NOVEMBER 2, 2011

Selecting Text

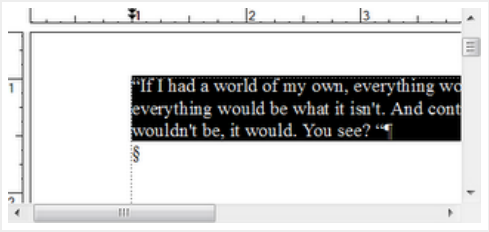
Once you understand text ranges, selecting text is easy. The following script selects the first paragraph in the active document.

Setting an insertion point is also easy; just make the beginning and ending text locations the same. (In other words, an insertion point is a text range and not a text location.)

Note the use of the constant `FV_OBJ_END_OFFSET` to determine the end of the paragraph. It includes all contents and the end of paragraph marker.

```
//get the first paragraph in the active document
var doc = app.ActiveDoc;
var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

var tRange= new TextRange(); //create a text range
//set the range to be the first paragraph
tRange.beg.obj = pgf;
tRange.beg.offset = 0;
tRange.end.obj = pgf;
tRange.end.offset = Constants.FV_OBJ_END_OFFSET;
//make the selection
doc.TextSelection = tRange;
```

Here is a paragraph (with a quote from Lewis Carroll), selected using the script.



Posted by Debra Herman at 7:24 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, NOVEMBER 3, 2011

Cut, Copy, Paste

Once you are able to make a selection in a document, you are able to use the same cut, copy, and paste commands an end user of FrameMaker has available. This turns out to be very powerful if you need to rearrange or remove text. For example, cutting and pasting a row in a table is far easier than deleting its components and then attempting to reconstruct them. The same is true if you need to rearrange elements in a structured document.

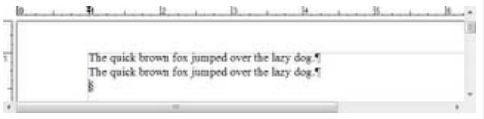
The `Cut()`, `Copy()`, and `Paste()` command all work with the current selection and each takes a single parameter: flags that indicate how the operation should proceed. A flag of `0` indicates that the operation should proceed without any interactive alerts or warnings.

A simple copy and paste example follows.

IMPORTANT: I took care to ensure that my test document has a second paragraph. In the real world, that condition might not hold. You need to verify that the paragraph exists before attempting the paste. I will discuss how to create a new paragraph in an upcoming past.

```
//get the first paragraph in the active document
var doc = app.ActiveDoc;
var mainflow = doc.MainFlowInDoc;
var tframe = mainflow.FirstTextFrameInFlow;
var pgf = tframe.FirstPgf;

var tRange= new TextRange(); //create a text range
//set the range to be the first paragraph
tRange.beg.obj = pgf;
tRange.beg.offset = 0;
tRange.end.obj = pgf;
tRange.end.offset = Constants.FV_OBJ_END_OFFSET;
doc.TextSelection = tRange;
doc.Copy(0); //copy the selected range
//select insertion point at the start of the next paragraph
tRange.beg.obj = pgf.NextPgfInFlow;
tRange.beg.offset = 0;
tRange.end = tRange.beg;
doc.TextSelection = tRange;
doc.Paste(0); //paste the text
```

Before paste



After paste

Posted by Debra Herman at 1:18 PM

2 comments:

frameexpert November 3, 2011 4:38 PM

If you use PushClipboard() before the cut or copy and PopClipboard() after the paste, you will preserve the original clipboard contents. Here is the description from the FDK docs for the FDK equivalent:

F_ApiPushClipboard() - Pushes the current Clipboard contents onto the Clipboard stack. It is useful if you want to use FDK Clipboard
functions, such as F_ApiCopy() or F_ApiCut(), without losing the Clipboard's original contents.

Reply

Debra Herman November 3, 2011 5:07 PM

That would be important if the script is being run by a user rather than as part of a batch process.

Thanks frameexpert!

Reply

Add comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

FRIDAY, NOVEMBER 4, 2011

Preventing Fireworks

A script that makes numerous changes to a document can create a flurry of screen activity that can be disturbing to a user. Successive cuts, copies, pastes, formatting changes, and the like can create "screen fireworks." Fortunately there is a way to save the user from viewing the changes as they happen: set the app property `Displaying` (session property in the FDK) to `false`.

You should also consider setting the app `Reformatting` property to `false`. This saves time as it tells FrameMaker not to reformat pages after each edit.

IMPORTANT: Don't forget to turn `Displaying` and `Reformatting` back to `true` when you are finished or you will leave the end user in the lurch!

When you are done, you need to explicitly reformat and then redisplay the document.

```
app.Displaying = false; //turn off screen redisplay
app.Reformatting = false; //turn off document reformatting

/*this does 100 pastes of the clipboard contents but any screen
intensive operation could appear here */
for (var i=0; i <100; i++)
    doc.Paste(0); //paste the text
app.Displaying = true; //turn back on
app.Reformatting = true;

doc.Reformat();//update the formatting in the document you changed
doc.Redisplay();//redisplay the document
```

Posted by Debra Herman at 11:02 AM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, NOVEMBER 5, 2011

A very important FDK include file

I have found that using the *Scripting Guide*, it is sometimes not easy (or maybe even possible) to map constant names to their numeric values.

You can easily find such information if you have a copy of the FDK include file `fapidefs.h`. To get it you need to download the FDK but I think you will find it is worth the trouble to do so.

NOTE: Go to http://www.adobe.com/devnet/framemaker.html for the download. Look in the `FDK10/include` directory for the file.

Here you will find such things as the values for error codes, hex values for Frame flags (`FF_*`), scripting values (`FS_*`), initializations (`FA_Init*`,) notificaitons (`FA_Note*`) and more. These will help you decipher values you see in the ExtendScript data browser. As the file is easy to scan, it may also give you a sense of what is possible in certain situations. Just be aware that the file may contain unsupported features. In addition, I have not verified that the FrameScript wrappers for the FDK support everything.

`fapidefs.h` is a code file intended to be viewed in Microsoft Visual C++ but, as it is a text file, you can view it in any editor that can read plain text including, of course, FrameMaker.

I include some snippets from this file below just to show you its format.

```
...
/*Settings for FA_errno */
#define FE_Success            0   /* All's well */
#define FE_Transport          -1  /* Communications is falling apart */
#define FE_BadDocId           -2   /* Illegal Document or Book */
#define FE_BadObjId           -3   /* Illegal Object */
...
/* Initializations */
#define FA_Init_First             1
#define FA_Init_Subsequent        2
#define FA_Init_TakeControl       3
#define FA_Init_DocReport         4

/* Notifications */
#define FA_Note_PreOpenDoc        1
#define FA_Note_PostOpenDoc       2
#define FA_Note_PreOpenMIF        3
#define FA_Note_PostOpenMIF       4
#define FA_Note_PreSaveDoc        5
#define FA_Note_PostSaveDoc       6
#define FA_Note_PreSaveMIF        7
...
```

Posted by Debra Herman at 8:20 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, NOVEMBER 6, 2011

Viewing Text Items

Use the `GetText()` method to work with the content of paragraphs or other objects that contain text. The method takes a single parameter that indicates the type of text items you are interested in. There is a long list of possibilities. See the *Scripting Guide* for full details. The method returns an array of text items.

The example that follows request text items in the main flow and request the following item types:

- Indicators of places where the character properties of the text change. (`FTI_CharPropsChange`)
- The identifier of the paragraph or text line to which the offset of each text items in the array is calculated. (`FTI_TextObjId`)
- The strings that make up the text. (`FTI_String`)

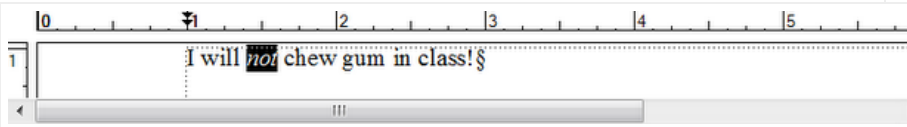
Why is it important to know the identifier of the object containing the text? While the code queries the flow, making any changes requires knowing the identifier of the paragraph (or text line) in which the text resides.

What about the text strings? If I run my script on a paragraph that has no changes to character properties will I get a single string? The answer is maybe. The FDK has a string limit size of 255 so no string will be larger than 255 characters. But, there is no guarantee that the text items will have the largest supported string size consistent with the information requested. In fact, if you call `GetText()` on a file that has been heavily edited, it is likely that you will get back a number of smaller text strings. There may be other reasons for this phenomenon that are not apparent to those not privy to the internal workings of the FrameMaker product. In short, `GetText()` gives you all of the text strings in their order of appearance but there is no guarantee that the number of text items is minimal.

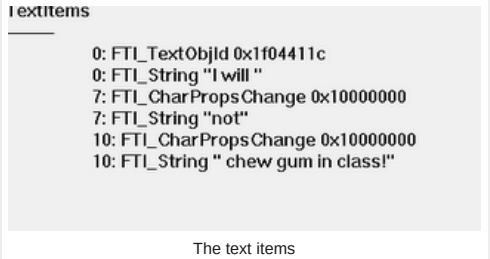
Imagine that GetText() is run, as shown in the code that follows, on the document below. The call is followed by a request that the text items found be displayed in the FrameMaker console.

NOTE: The constant values or OR'd to produce the appropriate flag parameter.

```
var tItems = mainflow.GetText(Constants.FTI_CharPropsChange |
Constants.FTI_TextObjId |
Constants.FTI_String);
PrintTextItems(tItems);
```



The file



The text items

The first number displayed is the offset from the paragraph start. In the case of `FTI_TextObjId`, you see the actual object identifier. In the case of `FTI_CharPropsChange`, the hex value indicates the type of change that took place. `FTI_String` items are followed by the actual string text.

Posted by Debra Herman at 12:03 PM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, NOVEMBER 9, 2011

Deallocating Text Items

The short answer is you don't need to if you are using the ESTK. Read on for the full saga.

The first words that came into my head when I learned about the ExtendKit toolkit for FrameMaker were *garbage collection*. Writing FDK programs is (mostly) a lot of fun but figuring out how to correctly allocate and deallocate memory is tricky and error prone. JavaScript purports to free one from those complexities.

So I was a bit surprised to see that the *Scripting Guide* documentation for `GetText()` and methods such as `GetProps()`, `Save()` and others contain notes that are similar to that shown below.

> "*Note: The returned TextItems structure references memory that is allocated by the method. Use the DeallocateTextItems() method to free this memory when you are done with using it.*"

So apparently memory management is still something script writers must contend with.

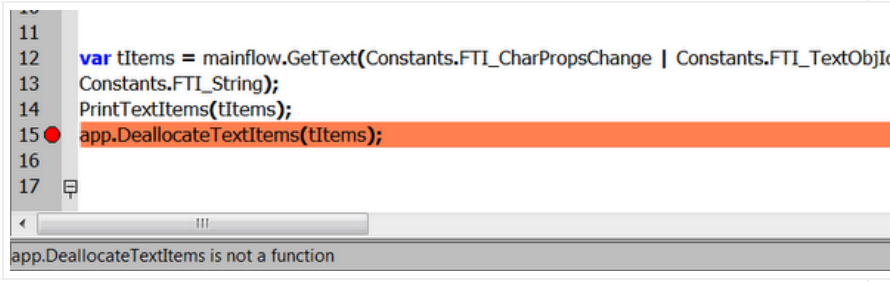
If you call `GetText()`, do you absolutely need to call the `DeallocateTextItems()` method to free the memory used when you are done? If you are *getting* a small number of text items, you are very likely safe if you forget or don't bother to deallocate. But if you call `GetText()` on a large quantity of text (possibly in many documents), failure to deallocate can be a problem. This is also the case if a script might be called many, many times before FrameMaker is restarted.

The FDK has the `F_ApiBailOut()` command which purports to direct a plug-in to stop running and give back its memory when it completes a command. I was never confident this did anything on Windows and there appears to be no analog in ExtendScript. Only Adobe knows how the scripts are implemented at that level. All one can be sure of is that when the user exits FrameMaker, any memory used by scripts is freed.

So, I tried to be a good citizen and deallocate my text items:

```
var tItems = mainflow.GetText(Constants.FTI_CharPropsChange |
Constants.FTI_TextObjId |
Constants.FTI_String);
PrintTextItems(tItems);
app.DeallocateTextItems(tItems);//WRONG
```

But I ran into a problem. I could not find a single example of the appropriate use of `DeallocateTextItems()`or the other deallocate methods anywhere. and my attempt to use this function produced the following error:



What is going on? Is this a documentation error or my error? I have yet to figure this one out.

Posted by Debra Herman at 7:32 PM

2 comments:

Ian Proudfoot November 10, 2011 12:37 PM

Debra,
The inclusion DeallocateTextItems() was a documentation error. This has been fixed in the latest version of the Scripting Guide.
http://help.adobe.com/en_US/framemaker/scripting/index.html

One way to double check this is to look at the ESTK data browser. This is a tree view of the available functions. With FrameMaker 10 running and the target application, select 'app' from the list. You will not find DeallocateTextItems().

By the way there are several undocumented functions in that list...

Reply

Debra Herman November 10, 2011 12:46 PM

Thanks again Ian. I did not realize I had an old version of the docs. I did look at the tree view and did not find deallocate functions but as the message was in the docs so many times I felt unsure as to whether or not I was missing something.

So good news, deallocation does not concern ESTK scripters.

Reply

Add comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

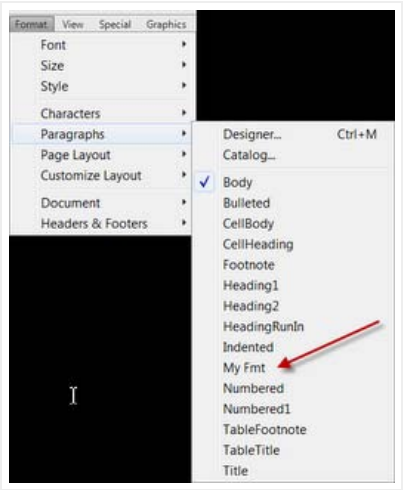
FRIDAY, NOVEMBER 11, 2011

Creating a new Paragraph Format

Paragraphs are, in FDK parlance, named objects. Use the `NewNamedObject()` to create one. Pass in the type of object and the name you wish to use.

```
var doc = app.ActiveDoc;
pgfFmt = doc.NewNamedObject (Constants.FO_PgfFmt, "My Fmt");
```

The newly created format has default properties.



You can create a long list of objects the same way including colors, condition formats, and character formats. (Look for the `Name` property.)

Posted by Debra Herman at 2:11 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, NOVEMBER 13, 2011

Updating Paragraph Format Properties Using Property List

Once you have created a paragraph format (or other named object), you very likely will want to modify the default properties automatically set when the object was created. You could do so one at a time but this can be tedious. It can also be hard to figure out just the right values for each property.

If your new format is similar to that of an existing format, you can use the properties of the existing format as a starting point. You can then explicitly set only those properties that are different.

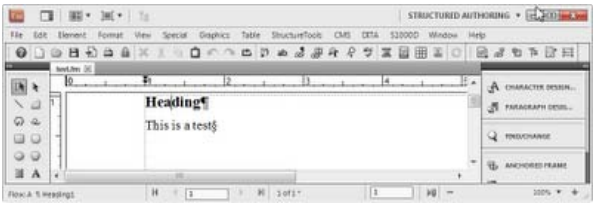
This example bases a new (or existing) format on an existing paragraph format. It changes only the underlining property.

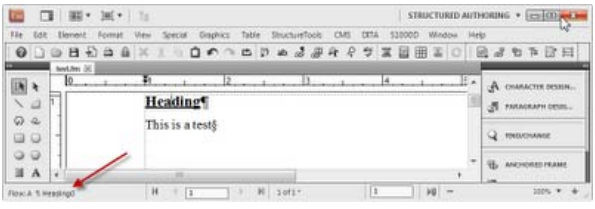
This script relies on the `GetProps()` and `SetProps()` methods which work with the properties of an object as a group.

NOTE: You can base the properties of a new paragraph format on those of a paragraph. This is the case even though the sets of properties do not exactly line up. Any mismatches are ignored.

The following function was called with:
`createPgfFmt(doc, "Heading0", "Heading1")`

```
/* Create or update a paragraph format with the name "newName". Set the
format properties based on those of the "basedOnName" format. Change the
underlining to be single.
*/
function createPgfFmt(doc, newName, basedOnName)
{
//determine if the format already exists
var newPgfFmt = doc.GetNamedObject(Constants.FO_PgfFmt, newName);
if (!newPgfFmt.ObjectValid()) {
//create it if necessary
var newPgfFmt = doc.NewNamedObject(Constants.FO_PgfFmt, newName);
if (!newPgfFmt.ObjectValid())
return(0);
}
//get the based on format
var existingPgfFmt = doc.GetNamedObject(Constants.FO_PgfFmt,
basedOnName);
if (!existingPgfFmt.ObjectValid())
return (0);
//Get all properties of existing format
var props = existingPgfFmt.GetProps();
//find underlining in the array of properties
var index = GetPropIndex(props, Constants.FP_Underlining);
//update the underling value
props[index].propVal.ival = Constants.FV_CB_SINGLE_UNDERLINE;
//update the new format's properties to match the old, save for
underlining
newPgfFmt.SetProps (props);
return(1);
}
```





Posted by Debra Herman at 12:38 PM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, NOVEMBER 13, 2011

Creating a New Paragraph

While paragraph formats have names, paragraphs do not. They fall into a class known as series objects. A series object is any object, other than graphic object, that occurs in an ordered list. Pages and book components are also series objects.

Call `NewSeriesObject()` to create a paragraph or other series object. As series objects are part of ordered lists,you need to specify the type of object to create and the identifier of its predecessor in the list.

If you want the new paragraph to appear at the start of the flow, specify the flow object for its predecessor. For other objects, specify `0`.

```
//add a paragraph at the start of the specified flow
doc.NewSeriesObject(Constants.FO_Pgf, flow);


//add a paragraph after the paragraph specified
doc.NewSeriesObject(Constants.FO_Pgf, pgf);
```

W9ACUGJMJ4Y9

Posted by Debra Herman at 7:27 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, NOVEMBER 15, 2011

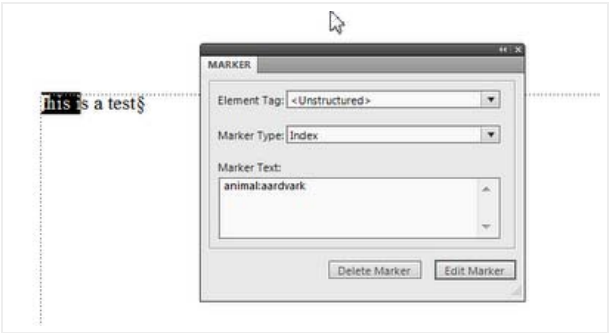
Creating an Index Marker

Markers are an instance of anchored objects. As anchored objects they are tied to a text location. Creating a marker means specifying a paragraph and an offset from the start of that paragraph. You also need to provide the same information a user might when creating such a marker. That means you must specify:

- The marker type
- The marker text

Use the same type name as appears in the user interface in the Marker Type pop-up menu. The marker text must also mirror that a user would enter.

The example below creates an index marker at the start of the first paragraph in the main flow.




```
var doc = app.ActiveDoc;
var flow = doc.MainFlowInDoc;
var tFrame = flow.FirstTextFrameInFlow;
var pgf = tFrame.FirstPgf;

function createMarker(doc, pgf, offset, type, text) {
    var tLoc, marker;
    tLoc = new TextLoc(pgf, offset);
    marker = doc.NewAnchoredObject(Constants.FO_Marker, tLoc);
    marker.MarkerType = type;
    marker.MarkerText = text;
    return 1;
}

createMarker(doc, pgf, 0, "Index", "animal:aardvark");
```

Posted by Debra Herman at 2:49 PM

6 comments:

Anonymous January 27, 2012 7:58 AM

Hi Debra,

thank you very much for your very helpful blog.

Did you ever try to insert another marker type than "Index", e.g. "Crossref"? I'm not sure, whether the line marker.MarkerType = "Index" is correct.

I want to insert some Crossref markers in my documents and I cannot find the right way - my markers are always of type "Index", no matter which "type" I use.

Best Regards, Andreas

Reply

Debra Herman January 27, 2012 11:03 AM

Hi Andreas,

I believe I made a mistake in this solution and got lucky as I chose Index Marker. I am working on a corrected solution. Stay tuned.

Debra

Reply

Debra Herman January 27, 2012 11:28 AM

Hi Andreas,

A some point the architecture for Frame markers changed to allow for user defined marker types. When I wrote this code, I forgot about the change. I am posting a new blog entry with the right code example.

Debra

Reply

Debra Herman January 27, 2012 12:33 PM

Hi again,

I have posted a correction in a new post dated Jan 27, 2012. Thanks for Andreas for finding this error.

Reply

Replies

Anonymous May 10, 2012 11:13 PM

Where can I find the corrected script. I'm trying to use it for cross-refs as well.
Thanks.
Michael

Anonymous May 10, 2012 11:17 PM

Nevermind my last post. I found it. :)

Thanks a lot.

Michael

Reply

Add comment

Comment as:

Extending FrameMaker

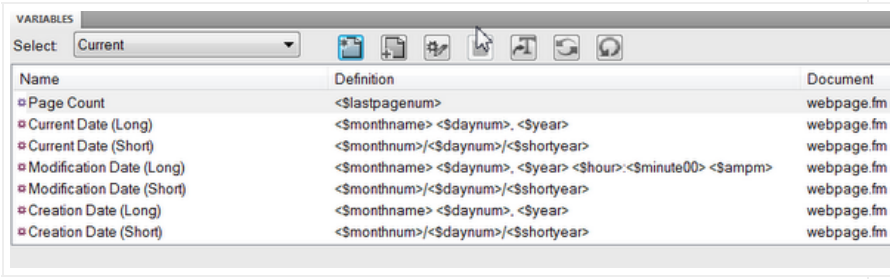
Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, NOVEMBER 16, 2011

Creating a Variable

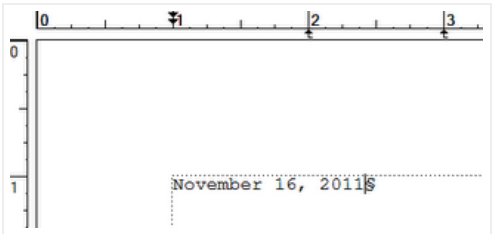
Variables fall into a category anchored formatted object. Tables and cross references also belong to this group. They are similar to anchored objects (markers) but creating them requires that you specify a format as well as an object type and text location.

In the case of variables, the format is simply the name it is know by in the user interface. You can view a list by using the `Special>Variables` command from the FrameMaker menu bar.



The following script adds the `Current Date (Long)` at the start of the first paragraph in the main flow.

```
var doc = app.ActiveDoc;
var flow = doc.MainFlowInDoc;
var tFrame = flow.FirstTextFrameInFlow;
var pgf = tFrame.FirstPgf;

function createVariable(doc, pgf, offset, type, format) {
    var tLoc, variable;
    tLoc = new TextLoc(pgf, offset);
    variable = doc.NewAnchoredFormattedObject(type, format, tLoc);
    return 1;
}

createVariable(doc, pgf, 0, Constants.FO_Var, "Current Date (Long)",
"Index");
```



Posted by Debra Herman at 2:03 PM

No comments:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, NOVEMBER 17, 2011

Page Types

FrameMaker employs four different page types. Three of these will be familiar to end users of FrameMaker:

- Body pages (`FO_BodyPage`)
- Master pages (`FO_MasterPage`)
- Reference pages (`FO_Reference Page`)

A fourth page type is the hidden page (FO_Hidden). There is only one such page and it holds hidden conditional text. While users cannot access this page, FDK plug-ins and scripts can.

There is no generic `FO_Page` object.

Body, master and reference pages can be made visible on the screen while hidden pages cannot be viewed in this manner.

NOTE: It is possible for a script to work with object on pages that are not currently displayed on screen.

Posted by Debra Herman at 12:01 AM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, NOVEMBER 17, 2011

Working with the Current Page

You can easily locate a document's current page using the `CurrentPage` document property. The current page is the one that is currently displayed on the screen and not the one with the insertion point.

Once you have the current page, you can access page number information:

- `PageNum` is an integer indicating the position of the page within the document. The count starts at zero.
- `PageNumString` is the number that appears in the page footer. It is defined by the variable building block <$curpagenum>.

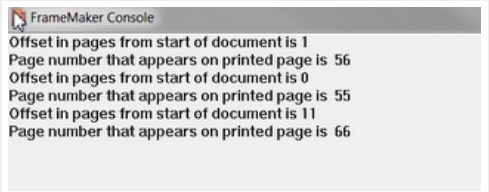
The script below defines two functions:

- `GetPageNumber()` returns the relative page number within the document.
- `GetPageString()` returns the portion of the printed page number determined by the <$curpagenum> building block. (In my experiments, redefining the `Current Page #` variable did not impact the value returned by this function.)

Each function is called on the active document. The string printed in the FrameMaker console reflects the values associated with the page currently on the screen.

```
var doc, pageNum, pageString;
doc = app.ActiveDoc;

function GetPageNumber(doc) {
    var page = doc.CurrentPage.PageNum;
    return page;
}

function GetPageString(doc) {
    var pageStr = doc.CurrentPage.PageNumString;
    return pageStr;
}

pageNum = GetPageNumber(doc);
pageString = GetPageString(doc);

Console("Offset in pages from start of document is " + pageNum);
Console("Page number that appears on printed page is  " + pageString);
```

The following output illustrates the fact that each of these functions likely returns a different value for the same document page even accounting for the difference in value type.



Posted by Debra Herman at 3:01 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, NOVEMBER 19, 2011

Navigating by Page

Moving through a document's pages is a straight-forward process. The first page, whether a body, master or reference, is a document property:

- `FirstBodyPageInDoc`
- `FirstMasterPageInDoc`
- `FirstRefPageInDoc`

As pages are ordered, you can also start at the end:

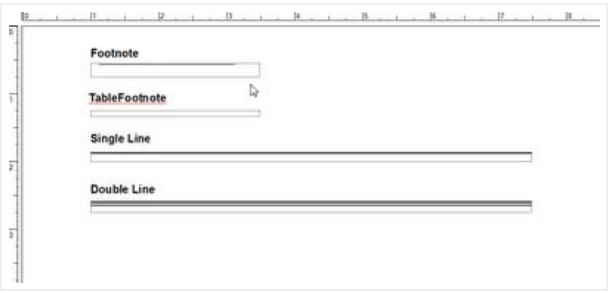
- `LastBodyPageInDoc`
- `LastMasterPageInDoc`
- `LastRefPageInDoc`

Each page has a `PageNext` and `PagePrev` property that allows you to move to the next page in the sequence. Remember that these are page properties.

Once you locate the page of interest, you can make it the current page. The following script makes the first reference page the active page.

```
var doc, firstRef;
doc = app.ActiveDoc;
firstRef = doc.FirstRefPageInDoc;
doc.CurrentPage = firstRef;
```

If you run this script on the `Portrait` template, the following page will appear on screen:



Posted by Debra Herman at 1:00 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

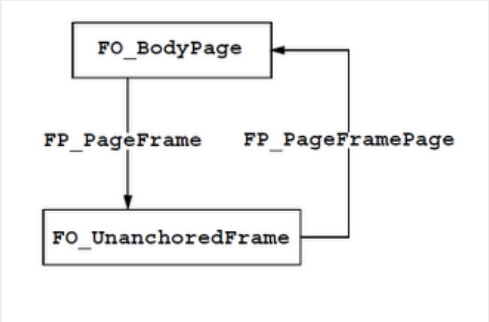
SUNDAY, NOVEMBER 20, 2011

The Page Frame

Pages object do not contain the text and graphics that appear on them. Instead, pages have a `PageFrame` property that specifies the identifier of the page frame that contains those objects. (Everything that prints in a FrameMaker document is within one page frame or another.)

The page frame is an invisible unanchored frame whose dimensions match that of the page. A page frame has the object type `FO_UnanchoredFrame`. If you have a page object, you can get to the page frame using the `PageFrame` property. There is a corresponding property `PageFramePage` that takes you from the page frame back to the page.

The following illustration shows this relationship. It uses FDK FP_ prefixes for property names as I have recycled it from some old FDK training materials. Drop them to get the corresponding ESTK names. All page types have this property, not just body pages.



These properties are important if you need to determine what page an object appears on or conversely what objects appear on a given page. I will go into these problems in more detail in my next several posts.

Posted by Debra Herman at 4:41 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, NOVEMBER 21, 2011

Graphics and their FrameParent

The page frame that holds all of a page's content is a special instance of a graphic object. But what exactly are graphic objects?

In FrameMaker, anything that has handles when selected is a graphic object. This includes the following object types:

- `AFrame`
  An anchored frame that is tied to a specific text location.
- `UnanchoredFrame`
  A frame that is placed directly on the page.
- `Line`, `Arc`, `Rectangle`, `Ellipse`, `RoundRect`, `Polyline`, `Polygon`
  The simple geometric shapes that can be created with the FrameMaker drawing tools
- `Group`
  An invisible object that allows a set of other graphic objects in the same frame to be moved or otherwise managed as a set.
- `TextLine`
  A line of text created with the FrameMaker drawing tools.
- `TextFrame`
  Container for a text flow.
- `Inset`
  An imported graphic or file that was imported into a FrameMaker document.
- `Math`
  An equation created with the FrameMaker equation editor.

All graphic objects appear within frames, whether anchored or unanchored. A graphic's `FrameParent` property provides the link to the containing frame. The page frame, as it has no containing frame has a `FrameParent` that is null.

Posted by Debra Herman at 2:56 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, NOVEMBER 22, 2011

Things that Reside in Text Frames

The previous post discussed the fact that graphic objects have a `FrameParent` property that returns their containing frame. Objects that can appear in text have a similar property that returns their containing text frame. These objects are markers, cross references, variables, anchored frames, and text insets of various types

While cross references, variables, and text insets contribute to the document content, markers do not. Anchored frames do not contribute to paragraph content although they can add considerable document content. For that reason, markers and anchored frames exist at a location in text while the other three have an associated text range that delineates the start and end of their content. Thus there are the following object/property pairings:

- Markers and anchored frames have the `TextLoc` property that returns an offset from the start of the containing table cell or paragraph.
- Cross references, variables, and the various text inset objects (`Ti*`) have the `TextRange` property that provides a starting and and ending text location.

Once you know a text location or range, you can easily get the identifier of the paragraph or table cell that starts (or ends) the text location. Those objects have the `InTextFrame` property that takes you to the (graphic object) text frame.

NOTE: Anchored frames have the `InTextFrame` property as well as the `TextLoc` property allowing you to go directly learn the idenfier of their text frame when desired.

Posted by Debra Herman at 7:43 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, NOVEMBER 23, 2011

Working with Selections

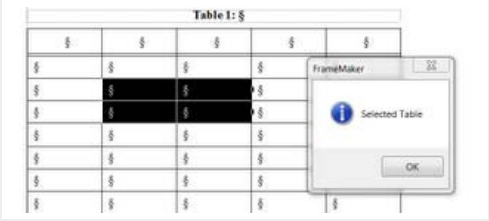
My plan is to write a script that determines the page location of the current user selection. This means figuring out what, if anything, in the document of interest is selected.

This script makes use of three document properties:

- `TextSelection` returns the text range for the current text selection.
- `SelectedTbl` returns the identifier of the selected table.
- `FirstSelectedGraphicInDoc` returns the identifier of the first in a list of selected graphics objects. `NextSelectedGraphicsInDoc`, a property of the first object found, takes you to the next object in the list.

Some things to note:

- If there is an insertion point or a selected text range within a table cell, that table is not selected. For a table to be selected, the entire table, the entire table title, or an entire cell within the table must be selected.



- The list of selected graphics in a document is an unordered list. This means that there is no telling which of a list of selected graphics might be returned by `FirstSelectedGraphicInDoc` . It might be the one the user selected first but do not count on that fact.

```
//returns a string indicating the type of object currently selected
function GetSelectedObjectType(doc) {
    var tRange, obj;
    tRange = doc.TextSelection;
    obj = tRange.beg.obj;
    if (obj.ObjectValid()) {
        return "Text Selection";
    }
    obj = doc.SelectedTbl;
    if (obj.ObjectValid()) {
        return "Selected Table";
    }
    obj = doc.FirstSelectedGraphicInDoc;
    if (obj.ObjectValid()) {
        return "Selected Graphic";
    }
    return "No selection";
}

var doc = app.ActiveDoc;
Alert(GetSelectedObjectType(doc), Constants.FF_ALERT_CONTINUE_NOTE);
```

Posted by Debra Herman at 6:11 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, NOVEMBER 28, 2011

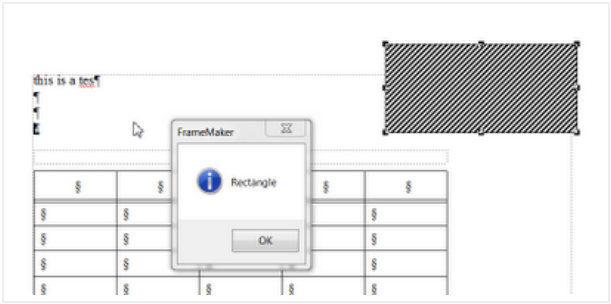
Determining an Object's Type

If you find an object by navigating a list such that the lists of paragraphs, markers, pages, and the like, you know in advance what object type you will find. But what if you get an object because it represents, for example, the user selection? In such a case, there is uncertainty as to what you have found but to do anything with that found object, you need to definitively determine its object's type.

The FDK provides the function `F_ApiGetObjectType()` which returns the `FO_` type of an object but there is no analog in ExtendScript. Your best option is to use the object's constructor property name. (Thanks to Ian Proudfoot for this tip.)

The following script updates the find selection type script to determine the object type of the currently selected object, if any.

```
function GetSelectedObjectType(doc) {
    var tRange, obj, type;

    tRange = doc.TextSelection;
    obj = tRange.beg.obj;
    if (!obj.ObjectValid()) {
        obj = doc.SelectedTbl;
        if (!obj.ObjectValid()) {
            obj = doc.FirstSelectedGraphicInDoc;
        }
    }

    if (obj.ObjectValid()) {
        type = obj.constructor.name;
    }
    else {
        type = "None";
    }

    return type;
}

var doc = app.ActiveDoc;
Alert(GetSelectedObjectType(doc), Constants.FF_ALERT_CONTINUE_NOTE);
```



Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, NOVEMBER 30, 2011

Determining an Object's Page

The script below works with the current selection. That selection can be an insertion point, a text range or a variety of objects including text frames, other frame types, graphic objects and equations.

It begins by determining the type of object selected. It handles, text selections, a selected table, or a selected graphic.

NOTE: If there is a text selection that spans paragraphs, the beginning of the selection is used in determining the object chosen.

If there is an object selected, the script then determines its page frame. It uses the page frame to determine the actual page number and displays it using an alert.

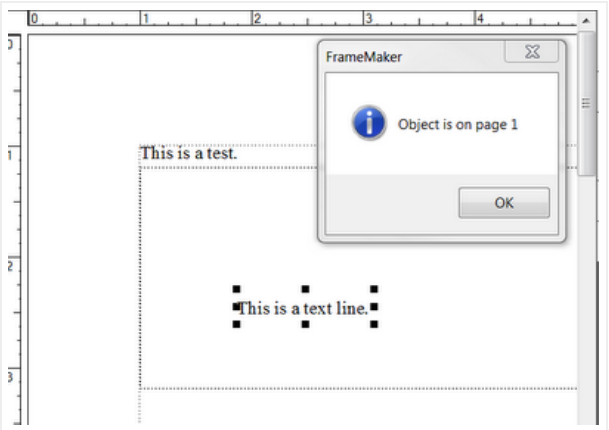
The real work of the script is in determining the page frame. To do so it works its way up the object chain until it finds an unanchored frame whose frame parent is null. The function contains a `while` loop that terminates only when the object found is a frame with no frame parent. For each object type, it moves up the chain of objects. If the object found is:

- A table, it uses the upper right most cell in the table.
- A cell, paragraph or anchored frame, it moves up the chain to the object's text frame.
- A text line, text frame, or a graphic shape, it moves up the chain to the object's frame parent.

The code is shown here followed by an example of its use.

```
function GetSelectedObject(doc) {
    var tRange, obj, type;
    obj =  null;
    tRange = doc.TextSelection;
    obj = tRange.beg.obj;
    if (!obj.ObjectValid()) {
        obj = doc.SelectedTbl;
        if (!obj.ObjectValid()) {
            obj = doc.FirstSelectedGraphicInDoc;
        }
    }
    return obj;
}

function FindPageFrame(doc, obj) {
    var frame, row, colNum, cell, objType;

    while (obj.ObjectValid()) {
        frame = obj;
        objType = obj.constructor.name;
        if (objType === "Tbl") {
            row = obj.TopRowSelection;
            colNum = row.RightColNum;
            cell = row.FirstCellInRow;
            obj = cell;
        }
        else if (objType === "Cell" || objType === "Pgf" ||
             objType === "AFrame") {
             obj = obj.InTextFrame;
         }
         else if (objType === "TextLine" ||  objType === "TextFrame" ||
                 objType === "UnanchoredFrame" || objType === "Arc" ||
                 objType === "Ellipse" || objType === "Group" ||
```

```
                objType === "Inset" || objType === "Line" ||
                objType === "Math" || objType === "Polygon" ||
                objType === "Polyline" || objType === "Rectangle" ||
                objType === "RoundRect") {
            obj = obj.FrameParent;
        }
    }//end while
    return frame;
}

var doc, frame, obj, page, pageNumStr;

doc = app.ActiveDoc;
obj = GetSelectedObject(doc);
if (obj.ObjectValid()) {
    frame = FindPageFrame(doc, obj);
    if (frame.ObjectValid()) {
        page = frame.PageFramePage;
        pageNumStr = page.PageNumString;
        Alert("Object is on page " + pageNumStr,
Constants.FF_ALERT_CONTINUE_NOTE);
    }
}
else
{
        Alert("No selection",  Constants.FF_ALERT_CONTINUE_NOTE);
 }
```



Selection is a text line

Posted by Debra Herman at 4:10 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, DECEMBER 1, 2011

Tables

Tables are represented by the `Tbl` object. They consist of one or more `Row` objects with each row having one or more `Cell` objects.

NOTE: Table cells are a special type of text frame. They can contain text and almost anything that can be inserted into text with the exception that tables cannot be inserted directly into table cells. You can add a table inside a table cell by placing an anchored frame within the cell, a text frame within that anchored frame and then inserting the table.

Tables have a large number of the properties that reflect the choices a user might make table design dialog box.

There are two ways to find tables:

- If you want all tables in any order, use the list of all tables in a document. Use the `FirstTblInDoc` document property, `NextTblInDoc` table property.
- If you want tables in flow order, use `GetText()` using the flag `Constants.FTI_TblAnchor`.

Posted by Debra Herman at 12:23 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

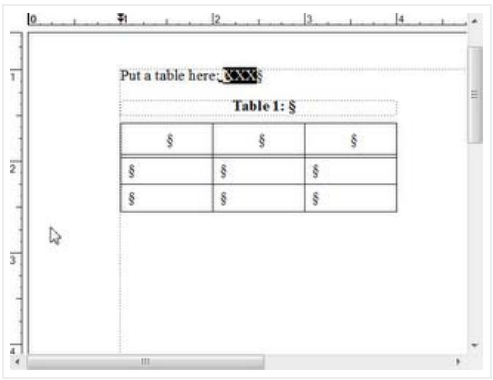
SATURDAY, DECEMBER 3, 2011

Creating a Table

Adding a table is a straight-forward task. You need to know the name of an existing table format, how many rows, columns, header rows, and footer rows to all and the location in text where the table is to be added.

The example below adds a table at the start of the current selection. It uses a table format found in the default `Portrait` template.

```
var doc, tRange, table;

doc = app.ActiveDoc;
tRange = doc.TextSelection;
table = doc.NewTable("Format A", 3, //number of rows
                                 2, //number of columns
                                 1,  //number of header rows
                                 0,  //number of footer rows
                                 tRange.beg);
```



Posted by Debra Herman at 1:42 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SUNDAY, DECEMBER 4, 2011

Adding Rows to a Table

Adding rows to a table requires that you know the identifier of the table and that of the row you want to add before or after. You then specify the direction you want to add in and the number of rows to add.

You can specify:

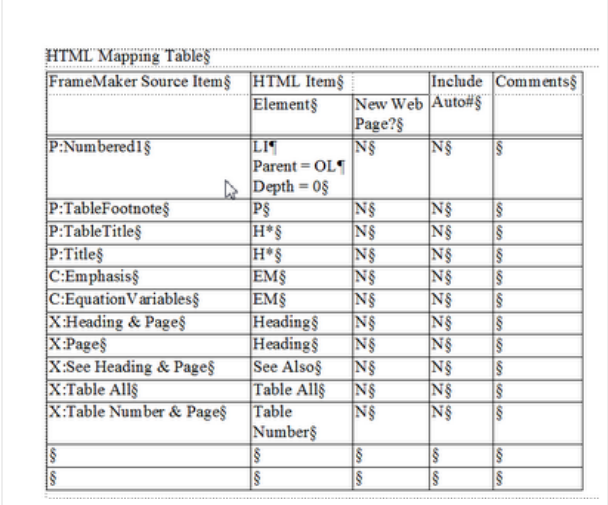
- Above (`FV_Above`)
- Below (`FV_Below`)
- To the Footer (`FV_Footing`)
- To the Header (`FV_Heading`)
- At the bottom of existing body rows (`FV_Body`)

The following example adds two rows below the last body row in the table. In the case of the Portrait template, this turns out to be the bottom of the HTML Mapping Table on the third reference page and not the first table on the first body page. This reflects the fact that the script uses `FirstTblInDoc`, an unordered list.

NOTE: To get tables in flow order use `GetText()` with the flag `FTI_TblAnchor`.

NOTE: Rows are accessed via an ordered list. They can be navigated top to bottom (`FirstRowInTbl`, then `NextRowInTbl`) or bottom to top (`LastRowInTbl`, then `PrevRowInTbl`).

```
doc = app.ActiveDoc;
table = doc.FirstTblInDoc;
row = table.LastRowInTbl;
row.AddRows(Constants.FV_Below, 2);
```



HTML Mapping Table§

FrameMaker Source Item§	HTML Item§		Include Auto#§	Comments§
	Element§	New Web Page?§		
P:Numbered1§	LI¶ Parent = OL¶ Depth = 0§	N§	N§	§
P:TableFootnote§	P§	N§	N§	§
P:TableTitle§	H*§	N§	N§	§
P:Title§	H*§	N§	N§	§
C:Emphasis§	EM§	N§	N§	§
C:EquationVariables§	EM§	N§	N§	§
X:Heading & Page§	Heading§	N§	N§	§
X:Page§	Heading§	N§	N§	§
X:See Heading & Page§	See Also§	N§	N§	§
X:Table All§	Table All§	N§	N§	§
X:Table Number & Page§	Table Number§	N§	N§	§
§	§	§	§	§
§	§	§	§	§

Posted by Debra Herman at 12:44 PM

No comments:

Extending FrameMaker

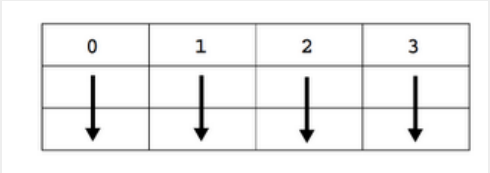
Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, DECEMBER 5, 2011

Adding Columns to a Table

Table columns differ from table rows because rows are object while columns are not. When adding a column you must specify the column number from which to start adding, the direction of the addition, and the number of coumns to add.

Columns are numbered from left to right with the first column being column 0.

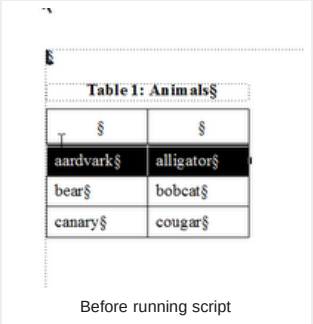


The direction is either `FV_Left` or `FV_Right`.

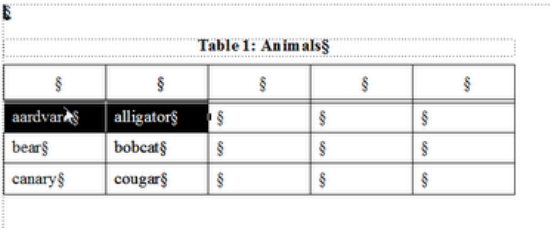
The following script adds three columns after the second column in the selected table.

NOTE: A table is selected if any of its cells are selected or the insertion point is in any of its cells.

```
var doc, table;

doc = app.ActiveDoc;
table = doc.SelectedTbl;
table.AddCols (1, Constants.FV_Right, 3);
```



Before running script



After running script

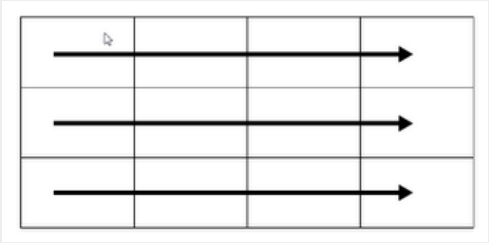
Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, DECEMBER 8, 2011

Getting Table Cells Row by Row

Once you have a table identifier, you can access each of the cells within that table. This post focuses on accessing cells from left to right and top to bottom.



To do so:

1. Get the first row in the table using the `Tbl` property `FirstRowInTbl`.
2. Get the first cell in that row using the `Row` property `FirstCellInRow`.
3. While there are additional cells, get each using the `Cell` property `NextCellInRow`.
4. While there are additional rows, get each using the `Row` property `NextRowInTbl`.

Here is a script that does just that and writes to each cell in the order visited.

```
var doc, table, row, cell, cellNum = 0;

doc = app.ActiveDoc;
table = doc.SelectedTbl;
row = table.FirstRowInTbl;
while (row.ObjectValid()) { //traverse rows
    cell = row.FirstCellInRow;
    while (cell.ObjectValid()) { //traverse cells in row
        cellNum++;
        var tLoc = new TextLoc();  //create text location object
        tLoc.obj = cell; //make it a cell
        tLoc.offset = 0; // insert at the start of the cell
        doc.AddText(tLoc, "Cell "  + cellNum);
        cell = cell.NextCellInRow;
    }
    row = row.NextRowInTbl;
}
```

`Note`: The first row in a table can be a header row (as shown) or a body row if no header exists.

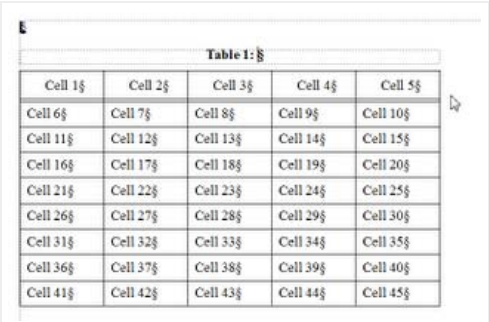


Table 1: §

Cell 1§	Cell 2§	Cell 3§	Cell 4§	Cell 5§
Cell 6§	Cell 7§	Cell 8§	Cell 9§	Cell 10§
Cell 11§	Cell 12§	Cell 13§	Cell 14§	Cell 15§
Cell 16§	Cell 17§	Cell 18§	Cell 19§	Cell 20§
Cell 21§	Cell 22§	Cell 23§	Cell 24§	Cell 25§
Cell 26§	Cell 27§	Cell 28§	Cell 29§	Cell 30§
Cell 31§	Cell 32§	Cell 33§	Cell 34§	Cell 35§
Cell 36§	Cell 37§	Cell 38§	Cell 39§	Cell 40§
Cell 41§	Cell 42§	Cell 43§	Cell 44§	Cell 45§

Posted by Debra Herman at 12:40 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

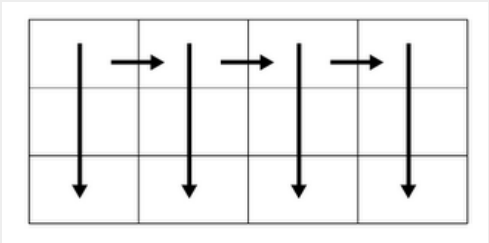
FRIDAY, DECEMBER 9, 2011

Getting Table Cells Column by Column

Traversing a table's cells column by column is a little tricky due to the fact that there is no column object. The following algorithm does the job:

1. Get the first row in the table using the `Tbl` property `FirstRowInTbl`.
2. Get the first cell in that row using the `Row` property `FirstCellInRow`.
3. While there are additional cells in the column, get the next cell in that column using the `Cell` property `CellBelowInCol`.
4. While there are additional cells in the first row, get the next cell in that row using the `NextCellInRow` property.
5. Go to step 3.

This cell traversal pattern is illustrated below.



The following script traverses the selected table using this algorithm.

```
var doc, table, row, topRowCell, cell, cellNum = 0;

doc = app.ActiveDoc;
table = doc.SelectedTbl;
row = table.FirstRowInTbl;
if (row.ObjectValid()) { //get first row
    topRowCell = row.FirstCellInRow;
    while (topRowCell.ObjectValid()) {//traverse cells in first row
        cell = topRowCell;
        while (cell.ObjectValid()) { //traverse cells in column
            cellNum = cellNum + 1;
            var tLoc = new TextLoc();  //create text location object
            tLoc.obj = cell; //make it a cell
            tLoc.offset = 0; // insert at the start of the cell
            doc.AddText(tLoc, "Cell "  + cellNum);
            cell = cell.CellBelowInCol;
        }
        topRowCell = topRowCell.NextCellInRow;
        cell = topRowCell;
    }
}
```

An example of the output produced by this script is shown below.

Table 1: §

Cell 1§	Cell 10§	Cell 19§	Cell 28§	Cell 37§
Cell 2§	Cell 11§	Cell 20§	Cell 29§	Cell 38§
Cell 3§	Cell 12§	Cell 21§	Cell 30§	Cell 39§
Cell 4§	Cell 13§	Cell 22§	Cell 31§	Cell 40§
Cell 5§	Cell 14§	Cell 23§	Cell 32§	Cell 41§
Cell 6§	Cell 15§	Cell 24§	Cell 33§	Cell 42§
Cell 7§	Cell 16§	Cell 25§	Cell 34§	Cell 43§
Cell 8§	Cell 17§	Cell 26§	Cell 35§	Cell 44§
Cell 9§	Cell 18§	Cell 27§	Cell 36§	Cell 45§

Posted by Debra Herman at 2:26 PM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

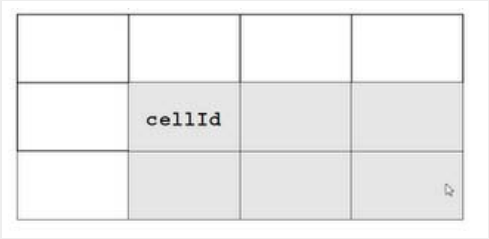
SATURDAY, DECEMBER 10, 2011

Straddling Table Cells

Table cells are said to be straddled if a rectangular group of cells is combined to form a single cell. When defining a straddle, you must identify the leftmost and uppermost cell and determine how many rows and columns are to be straddled.

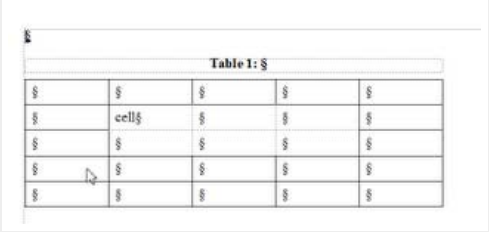
Straddles are determined by a number of cells to the right and below a specified cell. The FDK model is shown below with `cellId` referring to the second cell in the second row.

```
F_ApiStraddleCells(docId, cellId, 2, 3);
```



Here is the ExtendScript code and the result of straddling the second cell in the second table row.

```
var doc, table, row, cell;

doc = app.ActiveDoc;
table = doc.SelectedTbl;
if (table.ObjectValid()) {
    row = table.FirstRowInTbl;
    row = row.NextRowInTbl; //second row
    if (row.ObjectValid()) {
        cell = row.FirstCellInRow; //first cell
        cell = cell.NextCellInRow; // second cell
        if (cell.ObjectValid()) {
            cell.StraddleCells(2, 3); // 2 down, 3 across
        }
    }
}
```



WARNING: If this table had a header, the first row would be a header row. So don't get confused when counting rows in order to do a straddle. Header cells cannot be straddled with body cells.

Posted by Debra Herman at 3:00 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

TUESDAY, DECEMBER 13, 2011

Unstraddling Table Cells

Unstraddling table cells is, in some respects, the inverse of straddling table cells. It does remove the straddle but cell content originally dispersed between the straddled cells remains in the cell that "anchored" the straddle even after the original cells are restored.

The following code is identical to that shown in the previous post save for the addition of a single line:

```
cell.UnStraddleCells(2, 3);

var doc, table, row, cell;

doc = app.ActiveDoc;
table = doc.SelectedTbl;
if (table.ObjectValid()) {
    row = table.FirstRowInTbl;
    row = row.NextRowInTbl; //second row
    if (row.ObjectValid()) {
        cell = row.FirstCellInRow; //first cell
        cell = cell.NextCellInRow; // second cell
        if (cell.ObjectValid()) {
            cell.StraddleCells(2, 3); // create the straddle
            cell.UnStraddleCells(2, 3); // remove the straddle
        }
    }
}
```

Here is a table before the script is run:

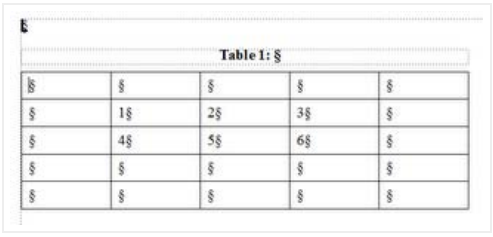


Table 1: §

§	§	§	§	§
§	1§	2§	3§	§
§	4§	5§	6§	§
§	§	§	§	§
§	§	§	§	§

Here is the same table after the second cell in the second row is straddled and then unstraddled.

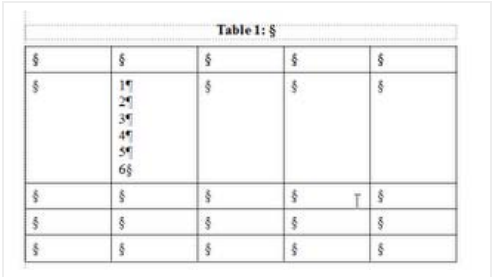


Table 1: §

§	§	§	§	§
§	1¶ 2¶ 3¶ 4¶ 5¶ 6§	§	§	§
§	§	§	§	§
§	§	§	§	§
§	§	§	§	§

Posted by Debra Herman at 1:50 PM

No comments:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

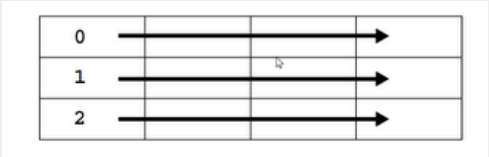
WEDNESDAY, DECEMBER 14, 2011

Making a Selection in a Table

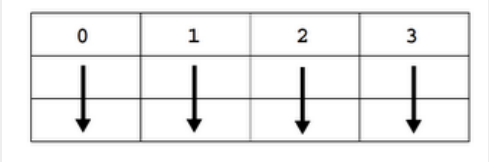
Table selections are powerful because they make it possible to use clipboard operations on a selected table region.

Table selections must be rectangular. Table selections work off of row and column numbers rather than row objects and column numbers as you might first expect.

Row numbering is from top to bottom with the first row numbered 0.



Column numbering is from left to right with the first column numbered 0.



The following script selects the first column in the currently selected table. Note the need to count the number of rows in the table before making the selection.

```
function countTableRows(table) {
    var count = 0, row;
    row = table.FirstRowInTbl;
    while (row.ObjectValid()) {
        count = count + 1;
        row = row.NextRowInTbl;
    }
    return count;
}

function selectLeftMostColumn(table) {
    var rowCount;
    rowCount = countTableRows(table);
    if (table.ObjectValid()) {
        table.MakeTblSelection(0, rowCount - 1, 0, 0);
    }
}

var doc, table;
doc = app.ActiveDoc;
table = doc.SelectedTbl;
selectLeftMostColumn(table);
```

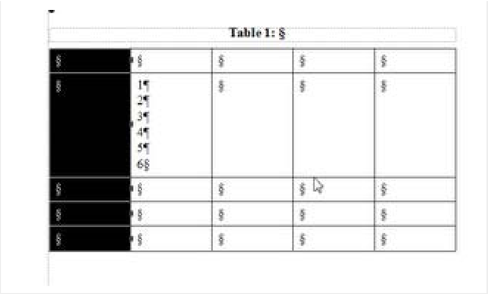


Table 1: §

§	§	§	§	§
§	1¶ 2¶ 3¶ 4¶ 5¶ 6§	§	§	§
§	§	§	§	§
§	§	§	§	§
§	§	§	§	§

Posted by Debra Herman at 12:01 AM

No comments:

Post a Comment

Comment as:

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

FRIDAY, DECEMBER 16, 2011

Structured FrameMaker: The Big Picture

Structure in a FrameMaker document is, in effect, an overlay of information. One can choose to work with a structured document as if there was no structure. In other words, you can work with paragraphs, tables, markers, and any other document content without regard to the fact that these may be wrapped in elements.

If, however, your goal is to manipulate XML content, you need to work with elements and their attributes and, possibly, element definitions. This means working with a new set of objects.

If you are working strictly with document content, you are working with:

- `Element`
- `Attributes`
- `Attribute`

If you want to alter or, more likely, query the Element Definition Document (EDD), you are working with:

- `ElementDef`
- `AttributeDefs`
- `AttributeDef`

If you are working with attribute expressions, you are working with:

- `AttrCondExpr`

If you need to manage the element catalog, you are working with:

- `ElementCatalogEntry`
- `ElementCatalogEntries`

If you want to manage text using elements rather than paragraphs, you are working not with `TextLoc` and `TextRange` but:

- `ElementLoc`
- `ElementRange`

Posted by Debra Herman at 12:16 PM

No comments:

Post a Comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

FRIDAY, DECEMBER 16, 2011

Determining if a Document is Structured

While it is common to speak of FrameMaker documents as having structure, only flows can be structured. In short, a FrameMaker document has structure if any of its flows are structured.

Typically, FrameMaker documents have a single structured flow and that flow is the main flow. (There can be special cases where it is helpful to have multiple structured flows but I am not going to address that possibility at this time.)

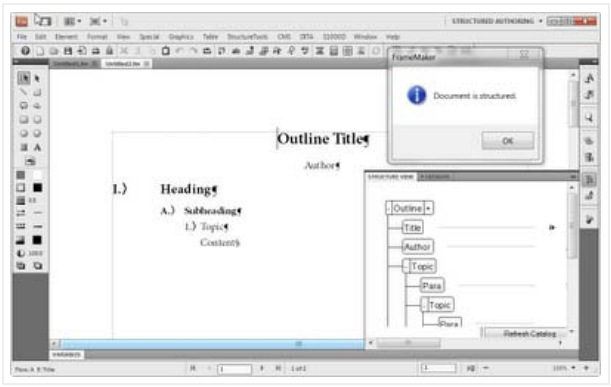
In the ordinary case, if you want to know if a document has structure, you simply need to get the main flow object and then determine if it has a highest level element.

The following script determines whether or not the main flow in the active document is structured.

NOTE: A document might have an element catalog but if its main flow lacks element tagging, this script considers it unstructured.

```
var doc, flow, root;

doc = app.ActiveDoc;
flow = doc.MainFlowInDoc;
root = flow.HighestLevelElement;

if (root.ObjectValid()) {
    Alert("Document is structured.", Constants.FF_ALERT_CONTINUE_NOTE);
} else {
    Alert("Document is unstructured.",
Constants.FF_ALERT_CONTINUE_NOTE);
}
```

The following example uses the built-in Harvard outline structured template.



Posted by Debra Herman at 12:30 PM

No comments:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

SATURDAY, DECEMBER 17, 2011

Getting an Element's Definition Name

When working with elements, it can be important to know an element's definition name. If you start with an element, you need to:

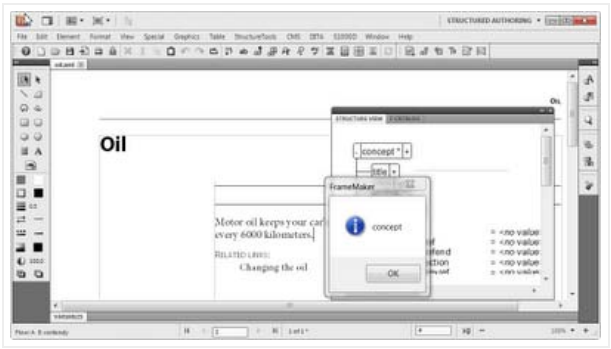
1. Get the associated element definition.
2. Get the definition name.

NOTE: A FrameMaker flow's element tree includes text nodes that do not have associated definitions and therefore have no name.

The following script displays the definition name of the highest level element in the main flow.

NOTE: The name returned is that in the EDD rather than in your DTD or schema.

```
function getElementName(elem) {
    var elemDef, name = null;

    elemDef = elem.ElementDef;
    if (elemDef.ObjectValid()) {
        name = elemDef.Name;
    }
    return name;
}

var doc, flow, root, name;

doc = app.ActiveDoc;
flow = doc.MainFlowInDoc;
root = flow.HighestLevelElement;

if (root.ObjectValid()) {
    name = getElementName(root);
    if (name !== null) {
        Alert(name, Constants.FF_ALERT_CONTINUE_NOTE);
    } else {
        Alert("Text node", Constants.FF_ALERT_CONTINUE_NOTE);
    }
}
```



Posted by Debra Herman at 1:01 PM

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, DECEMBER 19, 2011

Determining What Element is Selected (Part 1)

Element selections are a similar to text selections in some ways but differ in others.

Use the document property `ElementSelection` to get the current element selection.

Just as text selections consist of a *text range* with beginning and ending *text* locations, element selections are an *element range* with beginning and ending *element* locations.

The difference between the two emerges when you look at the element locations that define the starting and ending points for an element selection. Unlike a text location which is typically a paragraph and an offset from the start of that paragraph, an element location consists of:

- A parent element
- A child element
- An offset which is relative to the containing element.

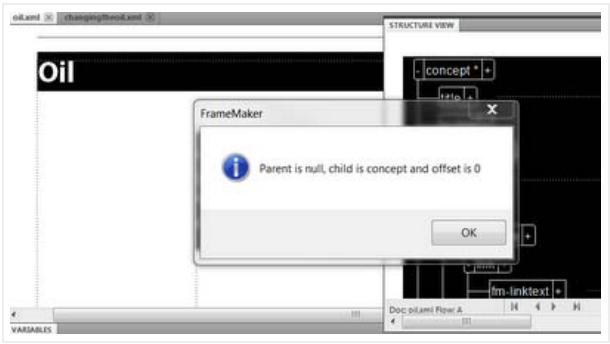
The following script demonstrates how to work with the beginning of an element selection. It can also be used as a tool for understanding the meaning of parent and child element in this context.

```
//Returns the element definition name
function getElementName(elem) {
    var elemDef, name = null;

    elemDef = elem.ElementDef;
    if (elemDef.ObjectValid()) {
        name = elemDef.Name;
    }
    return name;
}

//Returns location information for the current element selection
function getElementSelectionStart(doc) {
    var eLoc, eRange, locInfo;
    eRange = doc.ElementSelection;
    locInfo = {
        'parent' : eRange.beg.parent,
        'child' : eRange.beg.child,
        'offset' : eRange.beg.offset
    };
    return locInfo;
}

var doc, elem, pName = null, cName = null, locInfo;
doc = app.ActiveDoc;

locInfo = getElementSelectionStart(doc);

if (locInfo.parent.ObjectValid()) {
    pName = getElementName(locInfo.parent);
}
if (locInfo.child.ObjectValid()) {
    cName = getElementName(locInfo.child);
}
Alert("Parent is " + pName + ", child is " + cName + " and offset is "
+ locInfo.offset,
    Constants.FF_ALERT_CONTINUE_NOTE);
```

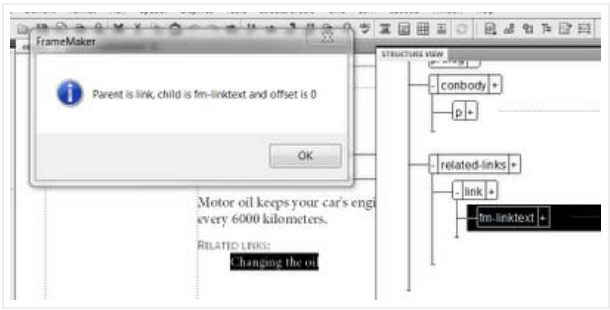
The first two test shown deal with the case where an entire element is selected. In these cases,

the offset from the start of the element is always 0.

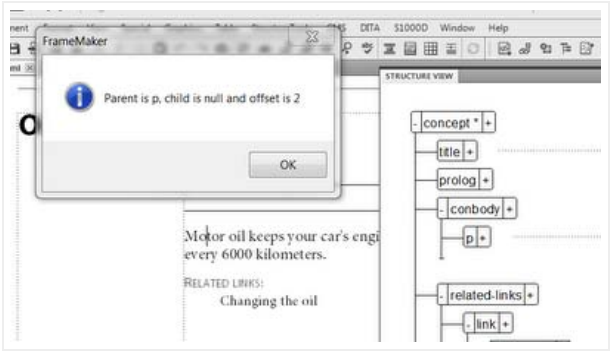
In the case where the root element is selected, the *parent* element is null and the *child* is the root element:



If an element other than the root element is selected, the *child* is the selected element and the *parent* is the parent of that child element.



This final example has an insertion point between the "o" and the "t" in "Motor" in the first document paragraph. In this case, the *parent* element is p (the paragraph element), and there is no child element. The *offset* from the start of p is 2.



Posted by Debra Herman at 7:16 PM

No comments:

Post a Comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

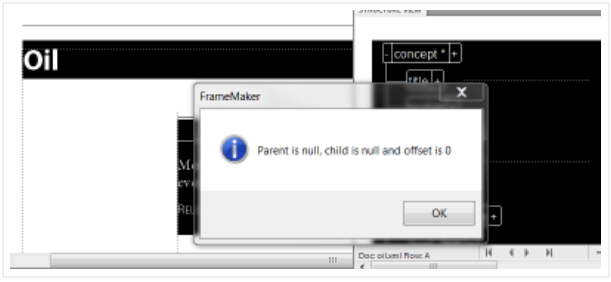
TUESDAY, DECEMBER 20, 2011

Determing What Element is Selected (Part 2)

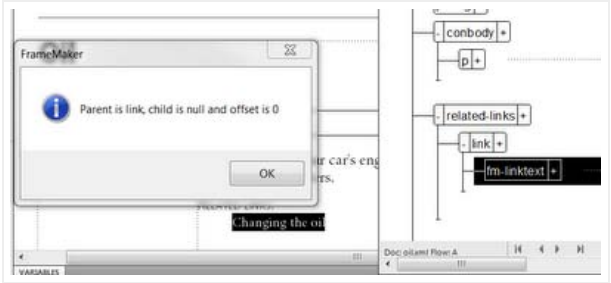
This post looks at the same element selections used in the previous post but this time, it is the end of the element selection that is of interest. Adapting `getElementSelectionStart()` is straight-forward as shown here:

```
function getElementSelectionEnd(doc) {
    var eLoc, eRange, locInfo;
    eRange = doc.ElementSelection;
    locInfo = {
        'parent' : eRange.end.parent,
        'child' : eRange.end.child,
        'offset' : eRange.end.offset
    };
    return locInfo;
}
```

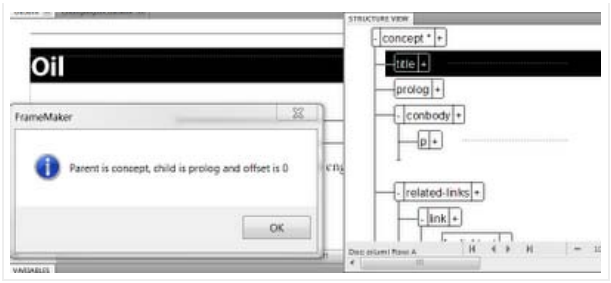
In the case where the root element is selected, the parent element end is `null` and the child is `null` and the offset is `0`. This makes sense as selecting the root element is equivalent to selecting the whole flow. There are no elements beyond that point.



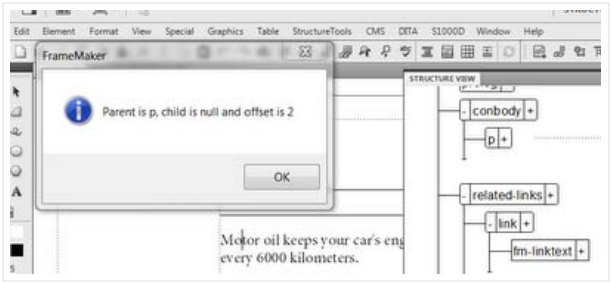
If an element other than the root element is selected and that element has no sibling following, the end *child* is `null` because its location is just beyond the last character in the selected element. The *parent* is the parent of the selected element.



In the case where the `title` element is selected there is a child element at the end of the selection as `title` has a "younger" sibling `prolog`.



This third example has an insertion point between the "o" and the "t" in "Motor" in the first document paragraph. In this case, the end *parent* element is p (the paragraph element), and there is no child element. The *offset* from the start of p is 2. This is exactly the same as was the case with the start element location as with an insertion point, as with an insertion point the starting and ending locations are the exactly same.



Posted by Debra Herman at 8:02 PM

2 comments:

Chris April 3, 2012 12:11 PM

Hi Debra.

Great blog. Question though: Would you recommend using the Doc.ElementSelection object as a way to ASSIGN a selection (i.e. to actually select)? I realize this is the other way around from using it to describe the user-selected selection but it seems necessary for what I want to do.

Thanks,

C Chew.

Reply

Replies

Debra Herman April 5, 2012 11:05 AM

Thanks for your comment.

You can make a selection. Basically you need to set up the desired ElementRange and then set the current selection to that range. I am working on an example. Look for it in a day or two.

Reply

Add comment

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

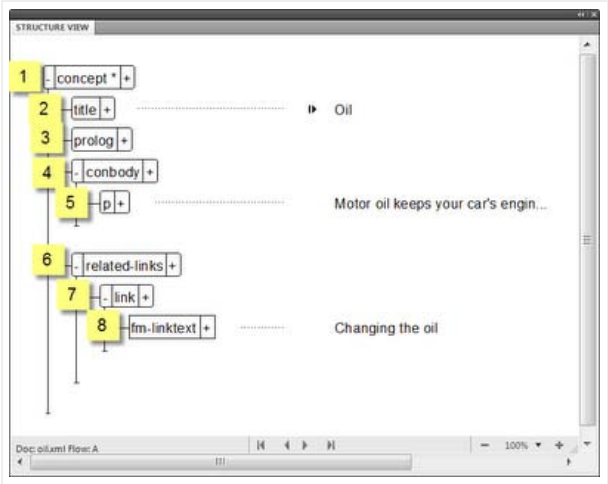
THURSDAY, DECEMBER 22, 2011

Preorder Element Tree Traversal

Elements differ from other objects in a FrameMaker document in that they are organized not into a list but into a tree. The tree structure is exactly that you would see in the document's `Structure View` window.

Typically, working with elements means starting with the highest level element in the structured flow, the root element, and visiting each of the elements in the tree in a predictable order.

The algorithm described below uses a preorder traversal which visits the example element tree shown below in the order marked.



The following script traverses the element tree using a recursive algorithm. The `walkTree()` function starts at the root (or whatever node is passed in). It gets that node's children starting with the first child. It is obtained using the element property `FirstChildElement`.

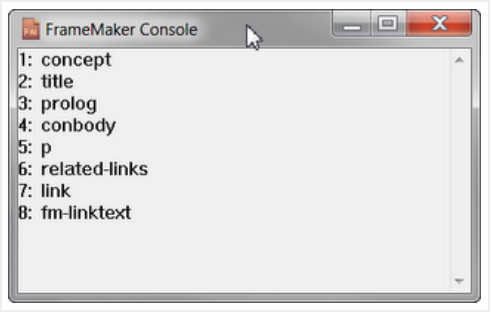
After processing that element, `walkTree()` calls itself this time passing in the child element. When that branch of the tree is processed, it continues on seeking any sibling elements using the element property `NextSiblingElement`.

NOTE: While not strictly necessary, the script keeps track of the element count to help make the association with the element numbers shown in the example above.

```
// Determines associated element definition name
function getElementName(elem) {
    var elemDef, name = null;

    elemDef = elem.ElementDef;
    if (elemDef.ObjectValid()) {
        name = elemDef.Name;
    }
    return name;
}

//Recursively traverses element tree in preorder fashion
function walkTree(elem, count) {
    var child, name;

    name = getElementName(elem);
```

```
    Console(count + ":  " + name);
    child = elem.FirstChildElement; //get first child if any
    while (child.ObjectValid()) {
        count = count + 1; //upate the element count to reflect element
found
        count = walkTree(child, count); //traverse subtree with child
root
        child = child.NextSiblingElement; //get the siblings
    }
    return count;
 }

var doc, flow, root;

doc = app.ActiveDoc;
flow = doc.MainFlowInDoc;
root = flow.HighestLevelElement;
if (root.ObjectValid()) {
    walkTree(root, 1);
}
```

The script output appears in the FrameMaker console and is shown here:



Posted by Debra Herman at 5:46 PM

1 comment:

Oliver John March 1, 2012 3:43 PM

Debra--

While this post is a couple of months old now, I wanted to come back to it and say thank you. This script opened my eyes to the power of recursive functions, and formed the basis of one of my most frequently used functions (building an array of elements)

Reply

Add comment

Comment as:

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

MONDAY, JANUARY 30, 2012

Text Item and Line Ends

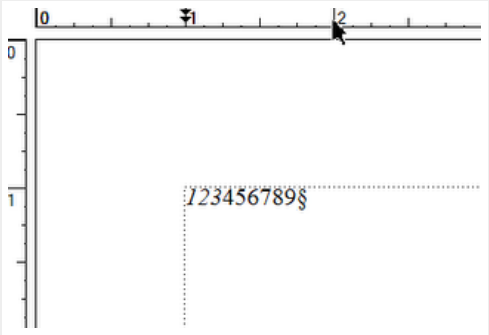
Understanding exactly what text items are produced when you call `GetText()` is critical to writing correct scripts. I have been looking at when you get a FTI_CharPropsChange notification upon calling `GetText()`. In particular, I am interested in what happens at line ends.

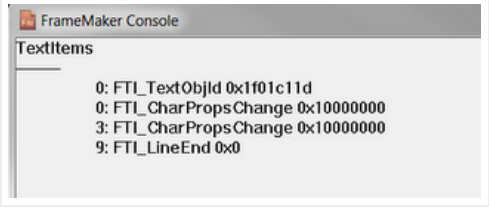
Each of the examples below show what happens when you run a simple script that prints a set of requested text items including changes in character properties and line ends to the `Console`. Each file tested has one "word" consisting of the digits from 0 through 9. In each case three of the digits have been italicized using the `Format>Style` menu.

I was expecting to see, among other things, an indicator of where any changes in the italic properties of the text begin and end. The problem I have found occurs when the italic text is at the end of a line. In such cases, I do not get the second FTI_CharPropsChange indicator I am expecting. This behavior deviates from what I knew to be the case in the FDK in earlier versions of FrameMaker. I have not yet tried this in the FM 10 FDK.

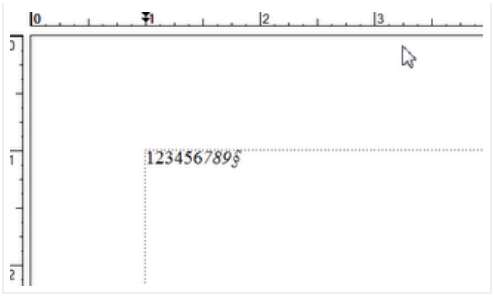
```
var doc, mainflow, tItems;

doc = app.ActiveDoc;
mainflow = doc.MainFlowInDoc;
tItems = mainflow.GetText(Constants.FTI_CharPropsChange |
Constants.FTI_TextObjId | Constants.FTI_LineEnd);
PrintTextItems(tItems);
```

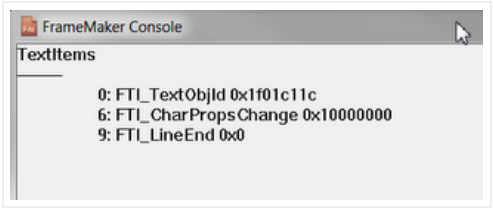
Here is a case that goes as expected, yielding a change indicator before and after the italicized text.



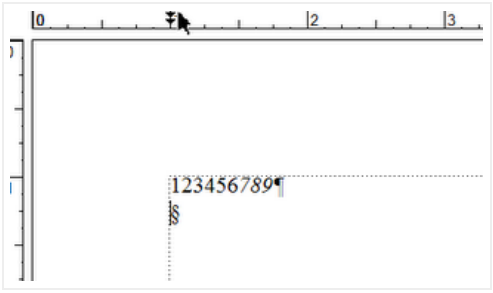


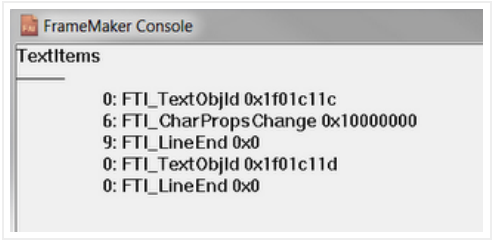
Now look at what happens when the italic text is at the end of the line. Here I get only the initial indicator. In my view, this is acceptable as this this a special end of flow case. There is no non-italic character after the italic "9".



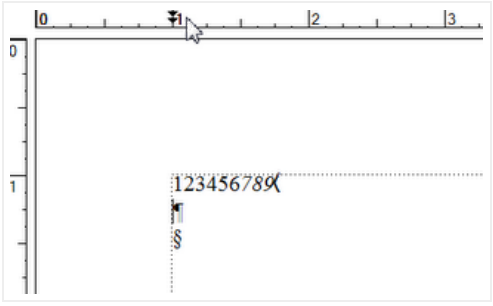


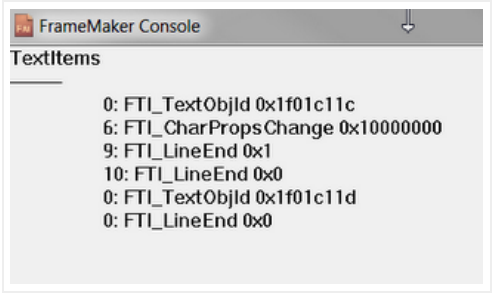
If a carriage return is added after the italic "9", I expect but do not get the second character properties change indicator. Note that end of paragraph symbols are selectable and this one is *not* italic.





Finally, lets use a soft return as our final character. Once again, there is only the beginning indicator.





I bring all this up not to be picky but because this change/bug complicates working with *GetText()*. If have I have missed something, please let me know.

Posted by Debra Herman at 7:13 PM No comments:

SUNDAY, JANUARY 29, 2012

Customizing a Save Operation

The following script uses the `Save()` method to alter the format in which a file is saved. Using this method requires more code than `SimpleSave()` but offers a whole range of customization options. It takes three pages in the *Scripting Guide* to document the possibilities. Whatever changes you want to make, however, can be accomplished by following the model shown here:

- Use `GetSaveDefaultParams()` to get a list of parameters whose default settings can be altered.
- Use `new PropVals()` to create an empty parameter list to be used to return information about the `Save()` operation.
- Use `GetPropIndex()` to locate one or more of these properties of interest.
- Reset that properties value in the parameter list.
- Call `Save()` passing in the save as name, and the two parameter lists.

The critical function is shown here:

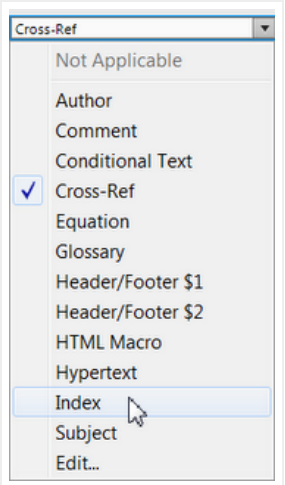
```
function saveAsPdf(doc) {
    var saveParams, name, i, baseName, status, fullName;
    name = doc.Name;
    baseName = dropSuffix(name);//drops everything from last period
    fullName = baseName + ".pdf";
    saveParams = GetSaveDefaultParams();
    returnParams = new PropVals();
    i = GetPropIndex(saveParams, Constants.FS_FileType);
    saveParams[i].propVal.ival =Constants.FV_SaveFmtPdf;
    doc.Save(fullName, saveParams, returnParams);
    i = GetPropIndex(returnParams, Constants.FS_SaveNativeError);
    status = returnParams[i].propVal.ival;
    if (status === Constants.FE_Success) {
        return (true);
    } else {
        return (false);
    }
}
```

Posted by Debra Herman at 8:37 PM No comments:

FRIDAY, JANUARY 27, 2012

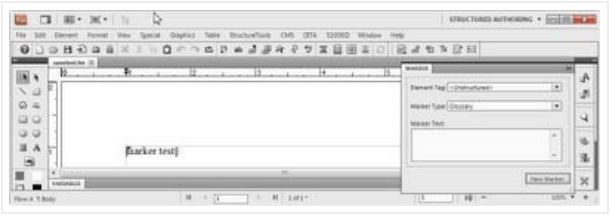
Creating a Marker (Correction to previous post)

When you create a marker you need to specify the marker type identifier. If you are using an built-in type, use the name found in the English user interface.



Use `GetNamedObject()` to get this identifier. Use the appropriate marker type name. (This is a document and not a session property because of the fact that new types can be added to a given document.)

```
function createMarker(doc, pgf, offset, type, text) {
    var tLoc, marker, markerType;
    tLoc = new TextLoc(pgf, offset);
    marker = doc.NewAnchoredObject(Constants.FO_Marker, tLoc);
    markerType = doc.GetNamedObject(Constants.FO_MarkerType, type);
    marker.MarkerTypeId = markerType;
    marker.MarkerText = text;
    return 1;
}
```



You only need to set the marker type properties for a built-in type if you want to change them. For example, add the following line of code to change the type name as displayed in the user interface.

```
markerType.Name = "My glossary";
```

Posted by Debra Herman at 11:47 AM 1 comment:

THURSDAY, JANUARY 26, 2012

Saving a document without user interaction

The following script makes a trivial change to that presented in my previous post to save the active document with no user interaction.

```
var doc, name;

doc = app.ActiveDoc;
name = doc.Name;
doc.SimpleSave(name, false); //no user involvement
```

Posted by Debra Herman at 2:17 PM No comments:

SUNDAY, JANUARY 22, 2012

Saving a document with user interaction

If your script makes changes to a document, you will want to save that file. This post, as will the next several posts, discuss the various ways you can do so.

From a scripting point of view, there are two possible ways to save a file. The first is deemed "simple" and the second is more complex to call and customizable. (In the FDK world, this option was deemed 'script-able.') The different between the two lies in the degree of configuration offered.

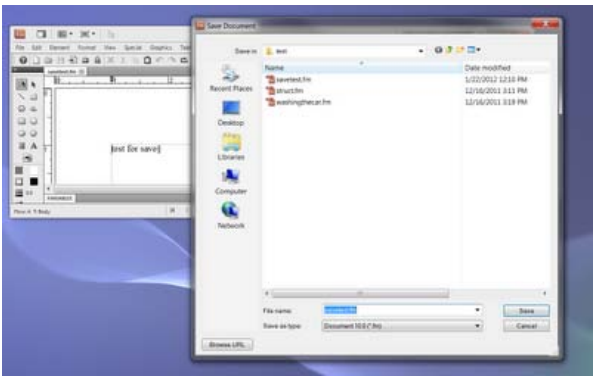
Simple operations use default settings but do offer one big choice. They allow you to choose whether you want to do the save operation with or without user interaction.

If you are doing a batch operation, you are not going to want user interaction. If, however, you are creating a command a user might call that needs user input, allowing a user to make choices regarding a save in the exact same way as he or she might had they invoked the command form the menu can be useful.

The following script invokes the save dialog using a script.

```
var doc, name;

doc = app.ActiveDoc;
name = doc.Name;
doc.SimpleSave(name, true); //true signifies user directed
```

Here is what happens when I saved a file using this script. It is the equivalent o the user choosing `File>Save`.



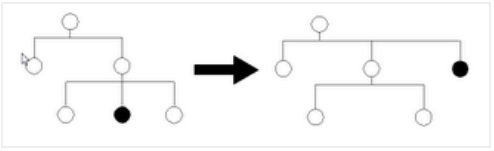
Posted by Debra Herman at 12:19 PM No comments:

WEDNESDAY, JANUARY 11, 2012

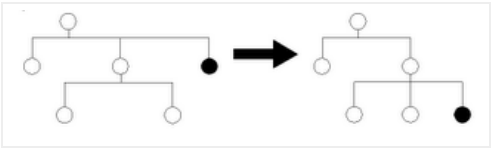
Rearranging the Tree of Elements

You can use the `Cut`, `Copy`, `Paste` methods to move selected elements around in the tree. There are, however, some special cases where you can take advantage of built-in tree manipulation functions.

- Use the document method `PromoteElement()` to make the selected element the sibling of its former parent.



- Use the document method `DemoteElement()` to make the selected element the child of its preceding sibling.



The following simple script promotes the selected element. You could easily modify it to demote an element.

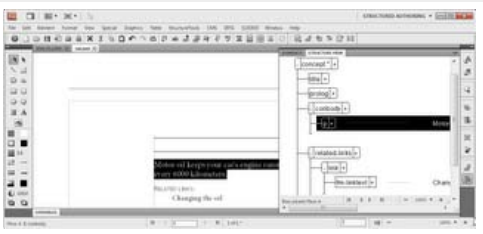
```
var doc;

doc = app.ActiveDoc;
doc.PromoteElement();
if (FA_errno !== 0) {
    Alert(FA_errno + '');
}
```

The script checks for an error code:

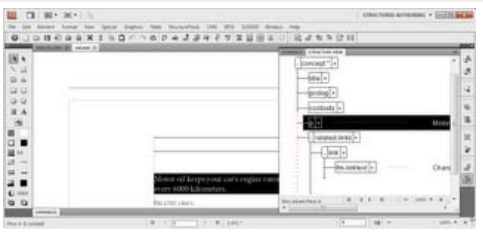
- `Constants.FE_WrongProduct (-60)` signifies that the product interface is not set to Structured FrameMaker .

- `Constants.FE_BadDocId (-2)` indicates a problem with the document identifier.
- `Constants.FE_BadSelectionForOperation (-59)` indicates that the action requested could not be taken on the selected element. For example, you cannot promote the root element or its children.

Here is the before and after that results from promoting the `p` element.



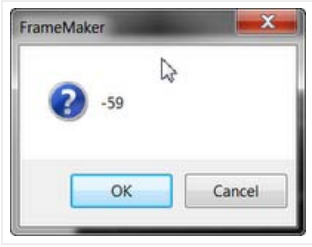
Before element promotion



After element promotion

This promote operation happens to make the structure tree invalid but the operation itself is sound.

Attempting to promote the root element produces the following error code message `Constants.FE_BadSelectionForOperation (-59)` as displayed by the script.



Posted by Debra Herman at 2:48 PM No comments:

SUNDAY, JANUARY 8, 2012

Text Nodes

Text nodes appear in the element tree when a container element has multiple children some of which are containers for text and some of which are just a text string.

Consider the case of an element, p, that allows text (<TEXT>) and other elements as described in the EDD:

Element (Container): p
General rule: (<TEXT> | dl | parml | fig | syntaxdiagram | imagemap | image | lines | lq | note | hazardstatement | object | ol | pre | codeblock | msgblock | screen | simpletable | sl | table | ul | boolean | cite | keyword | apiname | option | parmname | cmdname | msgnum | varname | wintitle | ph | b | i | sup | sub | tt | u | codeph | synph | filepath | msgph | systemoutput | userinput | menucascade | uicontrol | q | term | abbreviated-form | tm | fn-xref | xref | state | data | data-about | foreign | unknown | draft-comment | fn | indextermref | indexterm | required-cleanup)*

I have made a small modification to `oil.xml` to illustrate this point. The word "every" is italicized by wrapping it in the element `i`. Before this change, the `p` element had no children. Now it has three. The first and the last of these are text nodes as illustrated below.

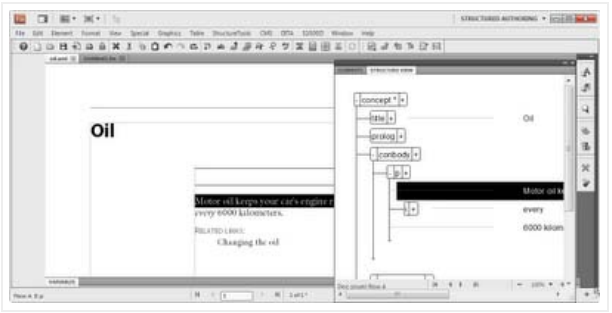
The function `getElementName()` uses the element property `ElementDef` to find the definition object and then the name property to get the name string.

```
function getElementName(elem) {
    var elemDef, name = null;

    elemDef = elem.ElementDef;
    if (elemDef.ObjectValid()) {
        name = elemDef.Name;
    }
    return name;
}
```

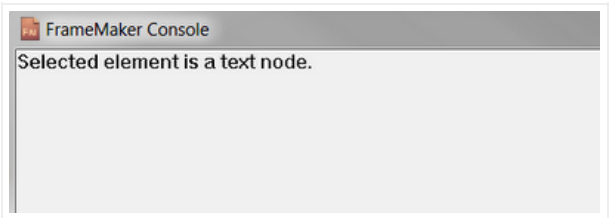
If an element definition is `NULL`, that element is a text node. Armed with this knowledge, the `displayAttrs()` function used in a previous post can be improved.

```
//writes element name and list of attributes to the console
function  displayAttrs(elem) {
    var i, name, aName, attrs;

    name = getElementName(elem);
    if (name != null) {
        Console(name);
        attrs = elem.GetAttributes();
        for (i = 0; i < attrs.len; i++) {
            aName = attrs[i].name;
            Console( "     " + aName);
        }
    } else {
        Console("Selected element is a text node");
     }
}
```

Here a text node is selected:



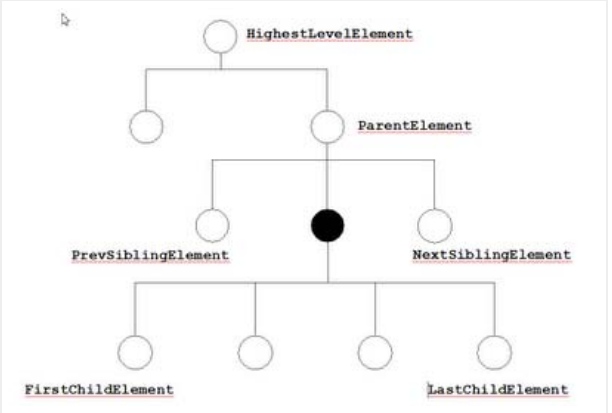
The output when `displayAttrs()` is called on the selected text node is shown here:



Posted by Debra Herman at 3:37 PM 5 comments:

Element Navigation Schematic

The following illustration shows the possible element properties for use in navigating the element tree in a structured document. All properties, with the exception of `HighestLevelElement` are those of the element shown in black.



Posted by Debra Herman at 3:02 PM No comments:

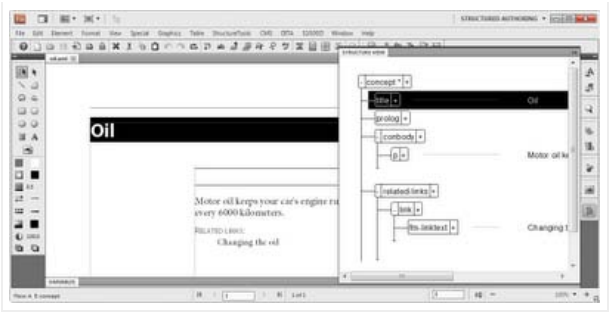
Getting Attribute Names

Once you have an element's object, you can examine its attributes. This script lists the attributes of the currently selected element in the FrameMaker console.

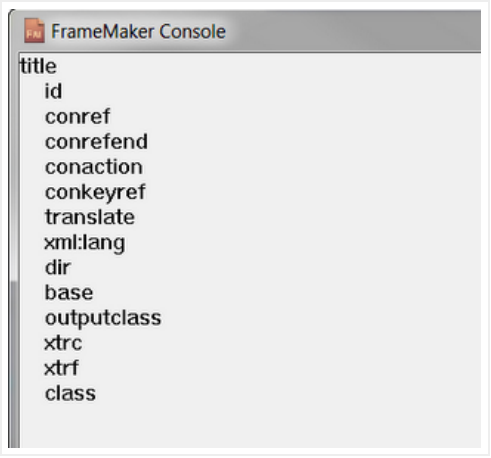
The script uses the `GetAttributes()` method and then iterates over the attribute list obtained, each time getting the name and writing that value to the FrameMaker Console. (The extra space prior to the name is used solely for readability purposes.)

```
// determines associated element definition name
function getElementName(elem) {
    var elemDef, name = null;

    elemDef = elem.ElementDef;
    if (elemDef.ObjectValid()) {
        name = elemDef.Name;
    }
    return name;
}

//writes element name and list of attributes to the console
function  displayAttrs(elem) {
    var i, name, aName, attrs;

    name = getElementName(elem);
    Console(name);
    attrs = elem.GetAttributes();
    for (i = 0; i < attrs.len; i++) {
        aName = attrs[i].name;
        Console( "     " + aName);
    }
}
var doc, elem, eLoc, eRange;

doc = app.ActiveDoc;
eRange = doc.ElementSelection;
eLoc = eRange.beg;
elem = eLoc.child;
if (elem.ObjectValid()) {
    displayAttrs(elem)
} else {
    Alert("No element was selected.",
        Constants.FF_ALERT_CONTINUE_NOTE);
}
```

Consider the test case where the `oil.xml` document's `title` element is selected.



The script produces the following output:



Posted by Debra Herman at 12:29 PM No comments:

MONDAY, JANUARY 2, 2012

Getting an Element's Text

Once you know how to work with text, working with an element's text is straight-forward. The following function calls the element `GetText()` method asking for strings. It then iterates over any text items found and concatenates the values found into a single string.

The `getElementText()` function is shown below. An example of its use will be provided in my next post.

```
function getElementText(elem) {
    var  tItems, i, text = "";

    tItems = elem.GetText(Constants.FTI_String);
    for (i = 0; i < tItems.len; i += 1) {
        switch (tItems[i].dataType) {
        case Constants.FTI_String:
            text =  text + tItems[i].sdata;
            break;
        }
    }
    return text;
}
```

Posted by Debra Herman at 9:18 PM No comments:

SUNDAY, JANUARY 1, 2012

Collapsing Elements in the Structure View

Once you can access an element's object, you can affect how it is displayed in the structure view.

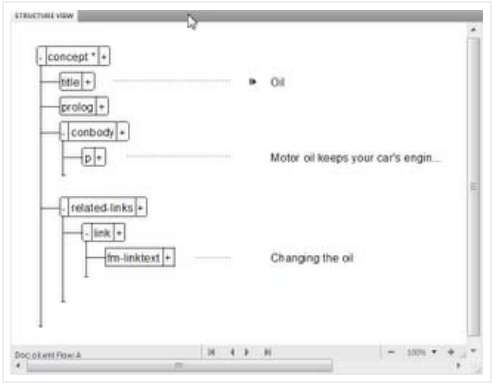
By updating the `walkTree()` function with a single line of code, your script can collapse, as is the case here, or uncollapse elements in the tree.

```
function walkTree(elem, count) {
    var child, name;

    name = getElementName(elem);
    elem.ElementIsCollapsed = true; // collapse element
    Console(count + ":  " + name);
    child = elem.FirstChildElement; //get first child if any
    while (child.ObjectValid()) {
        count = count + 1; //upate element count
        count = walkTree(child, count); //traverse subtree with child
root
        child = child.NextSiblingElement; //get the siblings
    }
    return count;
 }
```

Just be sure to redisplay the document when you are done to make your changes visible to the user.

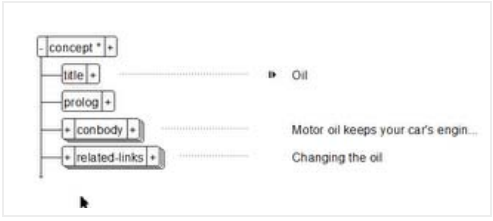
```
doc.Redisplay();
```

Before the script was run, the structure view appears as follows:



After the running the script, it appears as shown here:

Just to verify that all elements in the tree were collapsed, I manually uncollapsed the root element to reveal that the script works as expected.



Posted by Debra Herman at 6:35 PM No comments:

Newer Posts Home Older Posts

Subscribe to: Posts (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, FEBRUARY 22, 2012

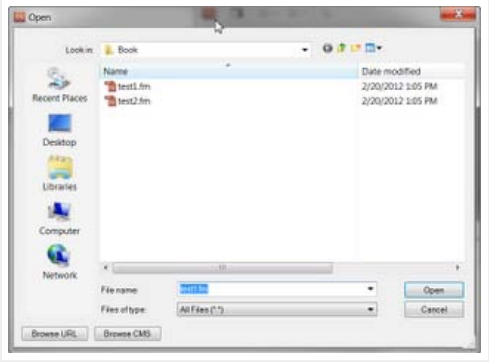
Opening a File with User Interaction

The easiest way to open a file is with `SimpleOpen()`. This is the analog to `SimpleSave()` discussed here.

If you do the open interactively, you do not need to know the file's location. You simply let the user locate the proper file.

The following script passes an empty string for the `fileName` parameter. FrameMaker assumes the file is the last one opened.

```
var doc;
doc = SimpleOpen("", true);
```



To open a file without user interaction, you need to pass it is full pathname and `false` for the second parameter.

Posted by Debra Herman at 2:40 PM 2 comments:

MONDAY, FEBRUARY 20, 2012

Working with Book Components

You might be tempted to think of FrameMaker books as containing documents. In fact, they contain book components which reference the files that when opened become FrameMaker documents.

To work with all of the documents in a book, you must traverse the list of book components and open each component in turn.

The list of book components is an ordered list. You can start with the `FirstComponentInBook`. Having found that component, you can use its `NextComponentInBook` property to find the next book component.

It is also possible to traverse a books components from last to first. In this case you start with `LastComponentInBook` and use the component's `PrevComponentInBook` property to move up the list.

The following code snippet shows how you to traverse a book's components from first to last.

```
component =book.FirstComponentInBook;
```

```
while(component.ObjectValid() ){
    doSomething();
    component =  component.NextComponentInBook;
}
```

Posted by Debra Herman at 1:21 PM No comments:

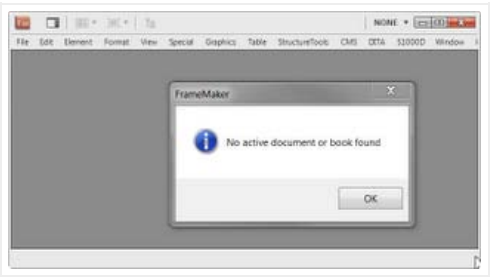
Working with the Active book

The `ActiveBook` is an exact analog of the `ActiveDoc`. It refers to the book that has the user focus.

There can be an active book *or* an active document but not both at the same time. It is also possible that there is neither an active book *nor* an active document.

When working with either an active book or an active document, you should check for the possibility that what you are seeking does not exist.

```
var doc, book;

book  = app.ActiveBook;
if (book.ObjectValid()) {
    Alert("Active book found", Constants.FF_ALERT_CONTINUE_NOTE);
} else {
    doc = app.ActiveDoc;
    if (doc.ObjectValid()) {
        Alert("Active document found",
Constants.FF_ALERT_CONTINUE_NOTE);
    } else {
         Alert("No active document or book found",
Constants.FF_ALERT_CONTINUE_NOTE);
    }
}
```



Posted by Debra Herman at 1:20 PM No comments:

SATURDAY, FEBRUARY 4, 2012

Converting Font Style Names to Font Angle Index Values

If your script queries a text property such as font angle, the value you get back is a number and not a name. For example, the following code snippet asks what is the font angle at the text location (`tLoc`) specified. That information is returned as an integer.

```
textProp =  doc.GetTextPropVal(tLoc, Constants.FP_FontAngle);
angleIndex = textProp.propVal.ival; //integer value
```

The integer value `angleIndex` is an index into an array containing the possible font angle values available in the current FrameMaker session. Use the app (session) property `FontAngleNames` to get this array of strings.

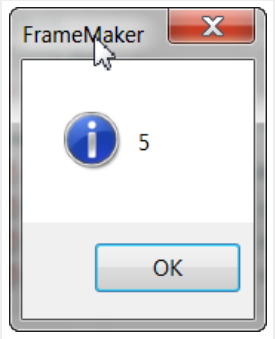
```
angleNames = app.FontAngleNames;
```

Here is that data structure as viewed in the ExtendScript DataBrowser:

The following script converts an angle name to its corresponding index in the `angleNames` array.

```
function findAngleIndex(angleName) {
    var angleNames, index;
    angleNames = app.FontAngleNames;
    for (index = 1; index < angleNames.len; index += 1) {
        if (angleNames[index] === angleName) {
            break;
        }
    }
    if (index === angleNames.len) {
        index = null;
    }
    return (index);
}

var index;

index = findAngleIndex("Italic");
Alert(index, Constants.FF_ALERT_CONTINUE_NOTE);
```

The output is as shown here:



Posted by Debra Herman at 8:34 PM No comments:

Newer Posts Home Older Posts

Subscribe to: Posts (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

WEDNESDAY, MARCH 14, 2012

Opening Files Off Screen

If you are processing a large number of files without user interaction, consider opening those files off screen. This allow your script to work faster (as there is no need to update the display) and to do its work without screen fireworks.

Use `Open()` and set the open parameter `Constants.FS_MakeVisible` to `False` to open a file off screen. Files opened this way can be modified in exactly the same way as files that are on screen.

If you work with files in this way, keep in mind is that your script must take total responsibility for managing off screen files. When a script is done with an off screen file, it must save (if changes are to be retained) and close the file. Failing to do so leaves the file locked and open but inaccessible to an end user.

There may be a case where you want to open a file off screen, work with that file, and then make it visible to users. To do so, set the document property, `FS_MakeVisible` to `True` when you are ready to reveal the file to users.

TUESDAY, MARCH 13, 2012

Opening Book Files when there are Issues

If you use `SimpleOpen()` to open book files non-interactively, files with missing graphics, missing fonts, or other issues will not open. Use `Open()` with an appropriate set of parameters to solve this problem.

The following function sets up open preferences that allow files of the following types to open without user interaction:

- Files which reference missing files.
- Files that are in an old FM version.
- Files with missing font issues.
- Files that are locked.

```
function getOpenPrefs() {
    var params, i;

    params = GetOpenDefaultParams();

    i = GetPropIndex(params, Constants.FS_RefFileNotFound);
    params[i].propVal.ival =
Constants.FV_AllowAllRefFilesUnFindable;
    i = GetPropIndex(params, Constants.FS_FileIsOldVersion);
    params[i].propVal.ival = Constants.FV_DoOK;
    i = GetPropIndex(params, Constants.FS_FontChangedMetric);
    params[i].propVal.ival = Constants.FV_DoOK;
    i = GetPropIndex(params, Constants.FS_FontNotFoundInCatalog);
    params[i].propVal.ival = Constants.FV_DoOK;
    i = GetPropIndex(params, Constants.FS_FontNotFoundInDoc);
    params[i].propVal.ival = Constants.FV_DoOK;
    i = GetPropIndex(params, Constants.FS_LockCantBeReset);
    params[i].propVal.ival = Constants.FV_DoOK;
```

```
    i = GetPropIndex(params, Constants.FS_FileIsInUse);
    params[i].propVal.ival = Constants.FV_OpenViewOnly;
    return (params);
}
```

The function to open the book using these parameters is shown here:

```
function openBookFiles(book) {
    var doc, component, compName;
    var openParams, openReturnParams;

    openParams = getOpenPrefs ();
    openReturnParams =  new PropVals();

    component =book.FirstComponentInBook;
    while(component.ObjectValid() ){
        compName = component.Name;
        doc = Open(compName, openParams, openReturnParams);
        component =  component.NextComponentInBook;
    }
}
```

Posted by Debra Herman at 1:13 PM No comments:

MONDAY, MARCH 12, 2012

Saving FM Binary in Old Version

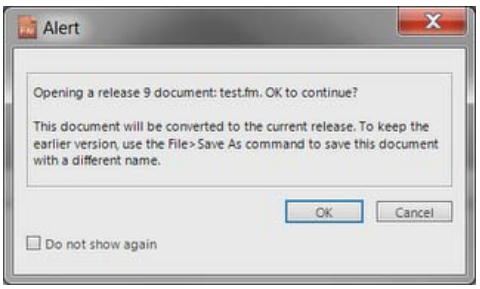
If you have ever needed to revert FrameMaker files to an older version, the following script does so automatically. It works on the active document but you could convert it to work on a book or a directory of files.

The key to the script is the `Save()` method which allows you to set the file type as desired. I have chosen FrameMaker 9 (`FV_SaveFmtBinary90`) but you can go as far back as FrameMaker 6 (`FV_SaveFmtBinary60`).

I chose not to change the file name. I end the script by closing the FM 10 version of the file without saving it. The FM9 version is already on disk.

```
var doc,  name, saveParams, i;

doc = app.ActiveDoc;
name = doc.Name;

saveParams = GetSaveDefaultParams();
returnParams = new PropVals();
i = GetPropIndex(saveParams, Constants.FS_FileType);
saveParams[i].propVal.ival =Constants.FV_SaveFmtBinary90;
doc.Save(name, saveParams, returnParams);
doc.Close(Constants.FF_CLOSE_MODIFIED);
```

After running the script, I opened the test file just to convince myself this really works.



Posted by Debra Herman at 3:30 PM 4 comments:

SUNDAY, MARCH 11, 2012

Using SimpleOpen() without User Interaction

`SimpleOpen()` can be called interactively or without user interaction. This distinction makes a difference if you are dealing with files that have issues.

If you are calling `SimpleOpen()` interactively, the user decides what to do when there are missing fonts, locked files, or other problematic situations.

If you call `SimpleOpen()` without user interaction FrameMaker defaults to very conservative behavior.
I created a simple a test to demonstrate this behavior. I cobbled together a set of files, each with a distinct issue, and put those files into a FrameMaker book. Each of my four files has one of the following issues:

- It is locked.
- It contains missing graphics.
- It uses unavailable fonts.
- It is in an older FrameMaker version.

Opening these files from the user interface brings up a dialog that allow the user to proceed or abort the open. Opening these files using the following code produces no user interaction and only one of the files opens: the locked file is opened in view only mode.

```
doc = SimpleOpen(compName, false);
```

Posted by Debra Herman at 3:17 PM 2 comments:

Newer Posts Home Older Posts

Subscribe to: Posts (Atom)

Extending FrameMaker

Learn how to customize Adobe FrameMaker with ExtendScript scripting.

THURSDAY, APRIL 5, 2012

Working with Structured Documents

When working with a structured document, you can choose to navigate the document using the tree of elements. Or, if it is more convenient, you can work with the document *as if* it does not have structure. That is, you can work with paragraphs in the main flow exactly as you would in unstructured FrameMaker.

Is there a downside to this approach? The answer is, it depends. If you are cutting and pasting or otherwise adding content, working outside the element structure may lead you to make changes that leave the structure invalid. But if you are simply inspecting the document or you are working with content that is not part of the element structure (e.g.. marker content), feel free to do whatever is easiest.

Posted by Debra Herman at 4:55 PM 2 comments:

Selecting an Entire Element

Selecting elements is tricky business. You will find helpful information in the *FDK Programmer's Guide.* I happen to have the one for version 6 on hand. In that version, you should refer to pages 366-7.

The key idea is to create an `ElementRange` that refers to the appropriate selection and then set the document's element selection to be that range. The following function sets the specified document's element selection to be the element passed in.

Element ranges have a beginning and an end. They consist of parent, child, and offset information.

The parent element is the containing element for both the beginning and end of the range. The starting child is the element itself. The ending child is the element's next sibling element. As the whole element is being selected, both offsets are zero.

```
//Selects the specified element in the specified document
function setElementSelection(doc, element) {
        var eRange = new ElementRange;

        eRange.beg.parent = element.ParentElement;
        eRange.beg.child = element;
        eRange.beg.offset = 0;
        eRange.end.parent = element.ParentElement;
        eRange.end.child = element.NextSiblingElement;
        eRange.end.offset = 0;
        doc.ElementSelection = eRange;
}
```

Posted by Debra Herman at 11:38 AM No comments:

Newer Posts Home Older Posts

Subscribe to: Posts (Atom)