West Street Consulting

FrameSLT 3.30 User Guide

Table of Contents

Chapter 1
Introduction to FrameSLT

Thank you for choosing to evaluate or purchase FrameSLT. At West Street Consulting, we are committed to providing products that serve real needs and helping you get the most out of them.

What is FrameSLT?

FrameSLT is a versatile, XPath-based processor that you can use to perform powerful queries and structural modifications within structured FrameMaker documents, without ever having to leave the FrameMaker interface. It includes the FrameSLT Node Wizard, which operates similarly to the FrameMaker Find/Replace dialog box, but far exceeds existing capabilities to search and manipulate your structured content. Additionally, it provides a flexible Node Wizard scripting feature that allows you to build complex and powerful routines for content manipulation and structural modification.

FrameSLT includes an exposed XPath parser and navigator that you can call with other API clients and third-party applications such as FrameScript® by Finite Matters Ltd®. Additionally, for FrameMaker 10 and later, you can call FrameSLT with ExtendScript. With the power of XPath, your custom applications can easily walk through a structure tree in ways not possible before without the addition of many lines of complex code.

You can use FrameSLT simply as a search tool, or you can use it to perform sweeping structural changes and content manipulation. Because FrameSLT has the ability to alter your structure and content, it is critically important that you read *"FrameSLT WARNING!"* on page 10 before using the product.

Who uses FrameSLT and when?

FrameSLT is indispensable for individuals who perform:

- Unstructured-to-structured conversions, especially with respect to the common "clean-up" stage following the main conversion stage.
- EDD maintenance and migrations. If you require a change to your structure definition(s), FrameSLT can sweep through dozens of pages of existing content to make the necessary changes, often in seconds. With FrameSLT, you are no longer shackled to an inadequate EDD just because you have so much content out there that currently adheres to it.
- Advanced content merging and manipulation. FrameSLT can pick through any structured file as if it were a database and extract the content for use in any other file. These capabilities include the ability to seamlessly move content between elements, attributes, markers, and more.
- Advanced authoring and/or publishing tasks that are not supported by native FrameMaker features, such as the generation of single-document TOCs.

FrameSLT may or may not be as relevant to the typical author on a daily basis. Its feature set is more catered to workflow architects; however, its capabilities are quire extensive and may find their way into many aspects of the authoring and publishing process.

Getting started with FrameSLT

Because FrameSLT is XPath-based, the Node Wizard is immediately ready for work on your structured documents, regardless of the EDD you are using. For the Node Wizard, getting started involves little more than understanding the basics of how the dialog box works. Node Wizard scripting is also immediately ready for work; however, the concepts involved can be complex and may require some experience to use effectively.

Note: FrameSLT includes a tutorial for the Node Wizard which may serve as an effective starting point for understanding the overall theme of the software. After a normal installation, this tutorial should be in the same folder as this document.

FrameSLT WARNING!

FrameSLT can perform sweeping, irreversible alterations to your structure and content. IT IS YOUR RESPONSIBILITY TO MAINTAIN THE INTEGRITY OF YOUR DATA. Before using FrameSLT to manage structure and content, you should be sure to have backups of all working files. In addition, after using FrameSLT, inspect your files carefully before saving the changes. A small XPath error can cause a major difference in the outcome.

If you keep backups and inspect your processed files carefully, your risk of data loss is low. In any case, however, West Street Consulting can not be held responsible for data loss that transpires as a result of FrameSLT usage, whether by user or application error.

Preferences

FrameSLT includes a set of preferences that affect various operations of the software. These settings are stored in a text file in the plugin installation folder or your “user profile” folder, depending on the initialization settings in your `FrameSLT.ini` file.

To open the file for editing in Notepad, you can select **FrameSLT > Open Preferences File**. At any time, you can select **FrameSLT > Read Preferences File** to refresh the plugin with current settings in the file.

Note the following:

- Descriptions and instructions for each setting are found within the settings file
- As you become more familiar with FrameSLT, you should review this entire file to be sure that the settings reflect the environment you want

Translation of the FrameSLT interface

FrameSLT supports customizable translations of its menus, dialog boxes, and messaging, based on “lookup” files that you can create and edit. When a string is required for a dialog box control or a message, it looks for that string in one of these lookup files according to the currently active language. Note the following:

- West Street does not claim support for any foreign language, only that you may add your own translations as desired. You can use this feature to implement a real language or simply rename labels, etc. using text that you like better. The plugin installs with a sample “Bogusian” language intended to serve as a model for setting up another language.
- West Street does not guarantee that any particular feature will work correctly once you implement a new language. We intend for it to work and will address any problems you find; however, you should be aware that it is impossible to fully test a feature with a virtually infinite number of variations/permutations.
- West Street believes that translation features cover about 95% of the strings that are associated with active features. This means that a small percentage of strings remain fixed in

English, especially as related to short prompts and other messaging. Additionally, note that none of the features scheduled for deprecation support translatable strings, such as the transformation features.

- West Street believes that the unicode range of character sets is fully supported for replacement text. The Bogusian sample provides an example of this.
- West Street believes that this feature is generally applicable for specialized use by select users only. For that reason, this documentation is brief. If you need assistance with translation features, please contact us and we will be happy to help.

Selecting a language

To select a language, select **FrameSLT > Language > Set Language**. Any languages that are properly configured will appear in the list (see *"Language configuration"* on page 11). A language change takes effect immediately.

You can also set a default language upon startup in your preferences file (see *"Preferences"* on page 10). This setting provides an option to default to the current language in use by the FrameMaker interface. Again, be aware that any setting in this file must represent a properly-configured language

Language configuration

For any new language, the plugin requires two lookup files, both of which much reside together in the plugin installation folder or the settings folder. These files are named as follows, where `*language*` is the case-insensitive language name that will appear in the **Set Language** dialog box:

- `FrameSLT_Strings_Dialogs_`*`language`*`.fm` - The lookup file for strings that appear in the menus and major dialog boxes, such as the Node Wizard. This file consists of a set of tables with the English text in the left column and the replacement text in the right. For each dialog box string, the plugin effectively starts with the English text and attempts to look up the translation based on the contents of this file.

  Note that for these types of strings, the plugin is starting with "built-in" English versions. Therefore, when set to English, the plugin does not use this file. That is, a file named `FrameSLT_Strings_Dialogs_English.fm` will never be used. However, it is always used for any other language.
- `FrameSLT_Strings_General_`*`language`*`.fm` - The lookup file for all other strings that strings that appear in error reports, short interactive prompts, and other places. The strings in this file are looked up based on an ID string, rather than the full English version. For this type of file, a `FrameSLT_Strings_General_English.fm` file does exist and is the source of all English strings that relate to prompts and messaging.

The two different files with their differing methodologies are required to accommodate how FrameMaker handles strings with respect to dialog boxes versus other functional areas, when programming to its API. Further explanation on this subject is beyond the scope of this document.

Once both of these files are properly-named and reside in the installation or settings folder, the respective language name automatically appears in the **Set Language** dialog box. Note the following:

- For new languages, the best approach is to copy the "Bogusian" examples and use them as templates. Each file contains additional instructions within.
- If you alter the file structure or otherwise make changes beyond that described in this document or within the files themselves, the results could be completely unpredictable. At worst, you may cause FrameMaker to crash.
- The strings files can also be stored in MIF format.

Additional language utilities

The plugin includes the following additional utilities in the **Language** menu that may be used rarely, if at all:

- **Create Dialog Strings File** - Creates a new dialog and menu strings file with English text only, ready for translation to a new language.
- **Update Dialog Strings File -** Attempts to update an active dialog and menu strings with the latest English strings used by the plugin. You must have a valid strings file currently open. Any new English strings are added as new rows to the respective tables. Any strings in the file that appear to be unused are colored red.

Note the following:

- These features were originally intended as a convenience for making updates, but may be deprecated. Again, it is recommended that you use the Bogusian files as templates instead.
- These features apply to the dialog strings file only. For the general strings file, you must always use an existing file as a template and all maintenance is done manually.

Chapter 2
About FrameSLT XPath

To use FrameSLT effectively, you must have a good working knowledge of XPath. You should review this information thoroughly before using FrameSLT, especially the details on which XPath components are supported and which are not. Nearly all FrameSLT functions rely on XPath to navigate the FrameMaker structure tree.

FrameSLT supports a subset of the W3C XPath standard. Supported components should behave exactly to standard. Use of non-supported components will likely cause parsing errors or unexpected query results.

Expansion of the FrameSLT-supported XPath is dependent on the needs of users like you. If you have a need for an XPath component that is currently not supported, we'd like to hear from you at info@weststreetconsulting.com.

About XPath

The XPath specification, defined by the W3C Consortium, allows querying and navigation within an XML-style structure tree. It is sometimes considered a simple language in itself and is frequently used during XML transformations to query source documents for content. Unlike a "linear" search, XPath allows you to find elements and attributes under very specific conditions, including considerations of structural hierarchy, positioning, and node content.

XPath is ideal for navigating a FrameMaker structure tree, because the markup of such a tree is very much analogous to XML markup. Without a language such as XPath, you would be limited to basic name and content searches provided by the standard FrameMaker Find tool.

There are a wealth of resources available for learning XPath, including the W3C website at www.w3.org and free tutorials at websites such as www.w3schools.com. Because so many options are available, this document does not attempt to reproduce a complete XPath reference here. However, you can get some beginners tips with *"XPath quick primer"* on page 13. And, you can see plenty of samples in *"FrameSLT XPath examples"* on page 28.

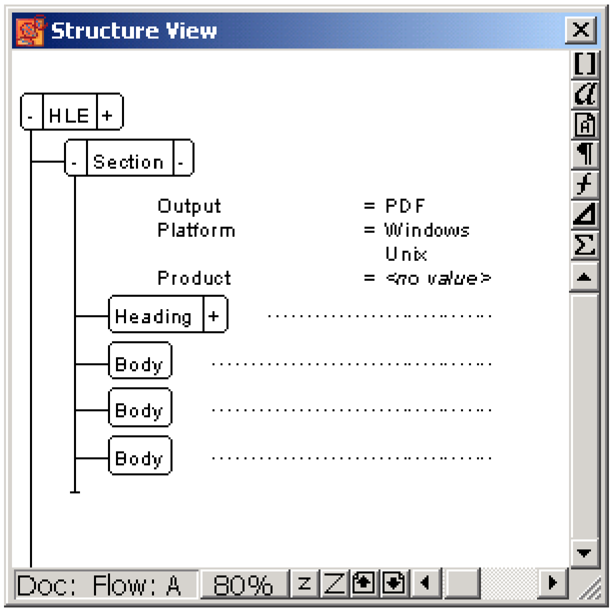
XPath quick primer

XPath is a special syntax designed for the express purpose of walking through a structure tree and finding very specific instances of elements, attributes, and other "nodes." It is reasonably simple to understand once you get started.

A node-matching expression is always a series of "axes" and "node tests." In essence, an axis tells which way to go, and the node test tells what to look for when you get there. For example, consider the following simple XPath:

```
child::Body
```

This expression says literally, "start at the context node (like an element), look to its children, and find any `Body` elements." Consider the following structure tree:



If the context element were the `Section` element, that XPath would find its three `Body` children. If the context were any other element, nothing would be found. In the FrameSLT Node Wizard, the currently selected element becomes the default context node. However, the selected element may not be relevant, if the first axis is a “go-to-root” axis, as explained in the next paragraph.

An important aspect of XPath is the first axis. In the previous example, the first axis (and only axis) is `child::` (go-to-child). So, a starting context must be manually provided (i.e., for the Node Wizard, the currently-selected element.) However, in many cases, especially with FrameSLT, you may find yourself using XPath that begins with the special “go-to-root” axis, indicated by a forward slash (/). This axis instructs the parser to begin at the root of the structure tree, using it as the initial context. With this axis, the context always starts at the root, and the currently-selected element is irrelevant.

As an example, the following XPath will find the highest-level element, `HLE`:

```
/child::HLE
```

It is very important to note that the forward slash *does not* set the `HLE` as the context... the context is actually “above” the `HLE`, at the true “root.” For example, the following XPath will find nothing, because the only child of the root is the highest level element, `HLE`:

```
/child::Section
```

However, the following expressions will find the `Section` element:

```
/child::HLE/child::Section
/descendant::Section
```

The descendant axis works because the `Section` is a descendant of the root. In fact, you can find any element by name with that particular expression. Note that the forward slash only means “go-to-root” if it is at the beginning. Otherwise, it is the delimiter between axis/node test components.

XPath also allows “predicates,” which are subexpressions in brackets used for testing something. You can use any axis in a predicate, and nest predicates within predicates as needed. For example, the following XPath will find the `Section` element again, because the predicate tests for the presence of an `Output` attribute:

```
/descendant::Section[attribute::Output]
```

In this case, the predicate doesn’t care what the value of `Output` is... only that the attribute exists. However, you can test values too, for example:

```
/descendant::Section[attribute::Platform = "Unix"]
```

That expression will find the `Section`, because the node test (`Section`) matches, and the predicate is satisfied. However, the following expression will find nothing, because the predicate is never satisfied:

```
/descendant::Section[attribute::Platform = "PDF"]
```

Once this begins to makes sense, take a look at the examples in *"FrameSLT XPath examples"* on page 28. Before long, you should be able to master XPath, and see just how versatile and powerful it is as a structure query tool.

Nodes vs. elements—Terminology

When discussing XML and XPath, the word "node" is used frequently to describe a generic location type within a structure tree. A node can be a place such as an element, an attribute, or a namespace... essentially any definable place in the structure tree that a query can step to. As you study XPath elsewhere, you will find this word used much more frequently than "element" and "attribute."

The FrameMaker interface and documentation, though, do not use this word, referring to locations specifically as elements and attributes. Therefore, the FrameSLT interface and documentation attempt to maintain this convention. However, when working with XPath, the word "node" is sometimes impossible to avoid, especially when the type of node is not specific. Therefore, an effort has been made in this document to adhere to the following terminology conventions:

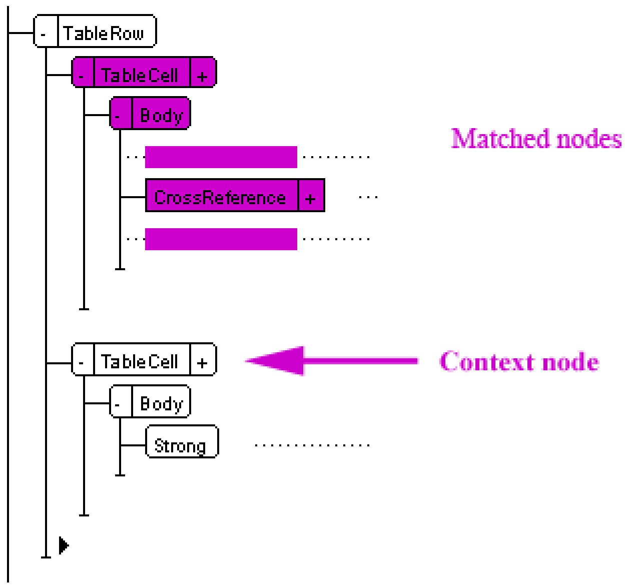
- **Node** When used alone, this word generally means "an element or attribute."
- **Element node** A FrameMaker element
- **Attribute node** A FrameMaker attribute

In reality, the term "node" refers more generally to any point within a branching structure where branches begin, terminate, or propagate. For the purposes of this document, however, an association with elements and attributes should be sufficient.

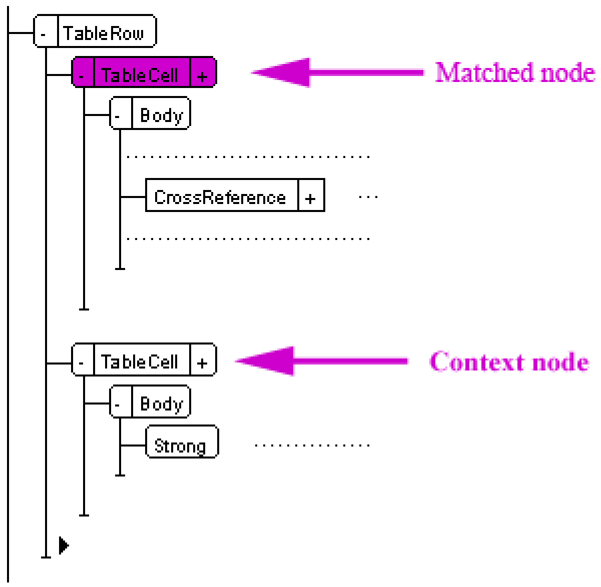
Supported axes

FrameSLT supports all standard XPath axes except `namespace::`. Using the "wildcard" character to indicate "any non-text node," the following examples illustrate supported axes:

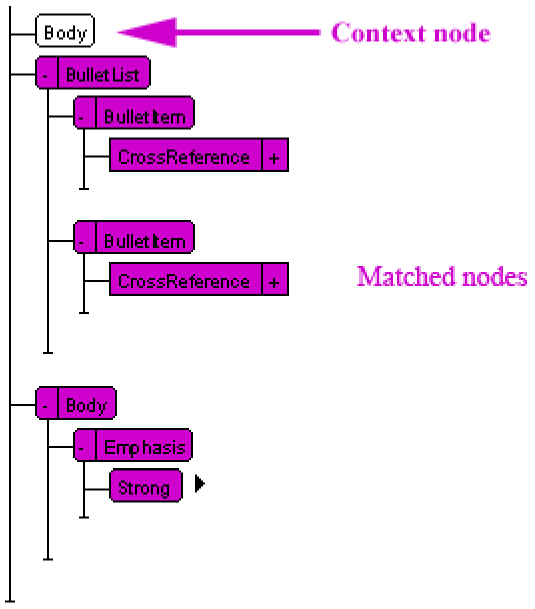
- `attribute::*`—Matches all attributes of the context node.
- `self::*`—Matches the context node.
- `child::*`—Matches all children of the context node.
- `descendant::*`—Matches all descendants (children, grandchildren, etc.) of the context node.
- `descendant-or-self::*`—Matches all descendants (children, grandchildren, etc.) of the context node, including the context node.
- `parent::*`—Matches the parent of the context node.
- `ancestor::*`—Matches all ancestors (parents, grandparents, etc.) of the context node
- `ancestor-or-self::*`—Matches all ancestors (parents, grandparents, etc.) of the context node, including the context node.
- `preceding::*`—Matches all preceding sibling nodes and all descendants of them, in document order. For example:



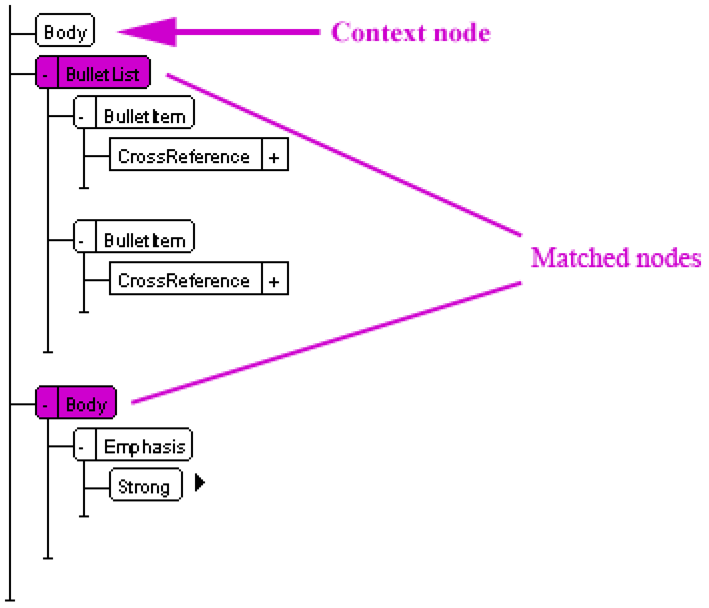
- `preceding-sibling::*`—Matches all preceding sibling nodes only, and excludes descendants, in document order. For example:



- `following::*`—Matches all following sibling nodes and all descendants of them, in document order. For example:



- `following-sibling::*`—Matches all following sibling nodes only, and excludes descendants, in document order. For example:



Special flow-related axes

FrameSLT implements the following set of non-standard axes that are designed for matching text frames, rather than markup nodes:

- `flow-body` - Match body page flows
- `flow-master` - Match master page flows
- `flow-ref` - Match reference page flows
- `flow-any` - Match any flow

These axes match the first text frame of the designated flow(s), after which other axes may be used as normal to drill further into the structure tree of the flow (if present). When using these axes, it is important to remember that they are matching text frames, not any content within those frames.

Note: Wildcards are not supported for the node test of flow-related axes. You must specify a valid flow name.

As an example, the following expression matches the first text frame(s) of all flows named “A” on the body pages of the document:

```
flow-body::A
```

The best way to further understand these axes is through experimentation and additional examples. For more information, see *“Flow-related queries”* on page 32.

Special “fmprop” axis

FrameSLT implements a non-standard `fmprop` axis for querying FrameMaker-specific object properties. The notation is similar to standard W3C-defined axes, but rather than indicating movement towards a node in the structure tree, it directs the retrieval of some property associated with the current element node (that is, the context node).

As an example, the following expression will match all elements that have the “Body” paragraph format applied to the underlying paragraph, or the first underlying paragraph if the element wraps multiple paragraphs:

```
//*[fmprop::PgfTag="Body"]
```

...where `PgfTag` is the specific notation that indicates a paragraph tag query. As another example, assuming that `Graphic` is a graphic element, the following expression matches all `Graphic` elements whose underlying anchored frame contains a referenced PNG file:

```
//Graphic[contains(fmprop::ImportObFile, ".png")]
```

Note that this evaluation would be case-sensitive, so a file with a `.PNG` extension would not make a match, unless the non-standard `contains-ci()` function were used instead. For more information, see *“Supported functions”* on page 24.

Currently, a very small subset of FrameMaker properties is supported by the `fmprop` axis, as described in the following table. There are many hundreds of potential properties available for evaluation, so it is not feasible to implement all of them at once. However, new properties will be added upon request. If you have a need to query a certain type of property, please contact us and we may be able to issue you a patch.

Additionally, note the following:

- These properties are only applicable for expressions that match elements within a text flow of a document. They are not applicable for expressions that match elements within a book structure tree or expressions that match text frames.
- The syntax of these properties follows the MIF tag format.

- Most of these properties can also be set using Node Wizard scripts.

`fmprop` property	What is retrieved

`FChangeBar`	Change bar status, either “true” or “false”. It can be used to find elements whose first paragraph (or parent paragraph, for text-range elements) is marked with a change bar. This setting is Boolean in nature and is only applicable with the “true” and “false” arguments. Examples:
	`//p[fmprop::FChangeBar="true"]`
	`//title[fmprop::FChangeBar="false"]`

`ImportObFile`	Full path of each imported (referenced) file in the underlying anchored frame. If the test matches a single file, the predicate is considered satisfied. This property is relevant to graphic elements only. Example: **`//Graphic[contains-ci(fmprop::ImportObFile, ".png")]`**
`MTypeName` **`MText`**	Marker-related properties of the underlying marker object, only applicable for marker elements. Any elements that are not markers are automatically disqualified by these tests. Further descriptions are as follows: • `MTypeName` - The marker type • `MText` - The marker text The following example matches all `IndexMarker` elements that use the “Index” type and contain the text “XPath”: **`//IndexMarker[fmprop::MTypeName="Index" and contains(fmprop::MText,"XPath")]`** In this example, if the `IndexMarker` element is not a marker element, the expression would never match anything, regardless of the text within the quoted strings.
`PageNumInt` **`PageNumStr`**	Page number(s) on which element content appears, as follows: • `PageNumInt` - The absolute page number, with the first page starting at 1. • `PageNumStr` - The “formatted” page number, as assigned in the document numbering properties. This value may or may not start with 1 and may or may not be an integer, according to the document numbering properties. For example, if numbering is set to use Roman numerals, this property may retrieve values such as i, ii, iii, iv, etc. Note that these properties will return multiple values if the element content spans multiple pages. Consider the following examples: **`//*[fmprop::PageNumInt = 1]`** ...matches any element with any content on the first page. **`//*[fmprop::PageNumInt > 1]`** ...matches any element with any content on any page after the first. **`//*[fmprop::PageNumInt = 1 and fmprop::PageNumInt = 2]`** ...matches any element with content that spans pages 1 and 2.
`PgfTag`	Paragraph tag assigned to the span of text that the element wraps. Example: **`//*[fmprop::PgfTag="Body"]`**

`fmprop` property	What is retrieved
`TblTag`	Table format tag of the current table, only applicable for table component elements. Any elements that are not table components are automatically disqualified by this test. Do not use this property to test paragraph container elements inside table cells; rather, use the `ancestor` axis to test the ancestor cell, row, or table instead. Example: **`//Table[fmprop::TblTag="Ruling"]`** In the previous example, if `Table` elements are not table components, the expression will never match anything, regardless of the text within the quotation marks:
`XRefName` **`XRefSrcText`** **`XRefSrcFile`**	Cross-reference-related properties of the underlying cross-reference object, only applicable for cross-reference elements. Any elements that are not cross-references are automatically disqualified by these tests. Further descriptions are as follows: • `XRefName` - The cross-reference format. • `XRefSrcText` - The reference ID; that is, the value of the “IDReference” attribute of cross-reference element. • `XRefSrcFile` - The source file that contains the destination of the cross-reference. If the cross-reference is internal to the document, the query returns an empty string. The following example matches all `xref` elements that use the “Heading on page” format and have destinations within the current document: **`//xref[fmprop::XRefName="Heading on page" and fmprop::XRefSrcFile=""]`** The following example matches all `xref` elements whose destination is located in the external file “`somefile.fm`”: **`//xref[contains(fmprop::XRefSrcFile,"somefile.fm")]`** In both examples, if the `xref` element is not a cross-reference element, the expressions would never match anything, regardless of the text within the quoted strings.

Special book- and file-related axes

FrameSLT implements the following non-standard axes that allow a query to traverse from documents to books and vice-versa, and directly from one file to another. For example, if you want to perform operations on a whole book using the Node Wizard or Node Wizard scripts, you would need to use one or more of these axes.

Note: The behavior of these axes can be difficult to understand. However, they are very important for advanced FrameSLT usage. If you need assistance with expression syntax, please contact West Street. For extended examples, see *"Cross-book and cross-file queries"* on page 31.

Axis	Behavior
`fmbook`	Matches: • The highest-level element (HLE) of the active book -or- • If no book is active, the HLE of the first book that can be associated with the active document -or- • Nothing, if no corresponding book can be found or no document is active at all This axis is the primary workhorse for stepping from a document tree into a book structure tree, noting that it goes straight to the book HLE and does not consider any current context. Element names are currently not considered, so the node test should always be simply an asterisk (*). For example, the following simple expression is valid and matches a book HLE: **`fmbook::*`** This expression will match the book HLE if the book is active or any of its chapters are active. Again, note that the current element selection or insertion point location is not relevant.
`fmcomp`	Matches the component-level element in book structure tree for the currently-active document. That is, it searches for an open book that contains the currently-active document as a chapter, then matches the respective component element in book structure tree. If the context is already a book structure tree, it matches nothing. This axis is an alternative for moving from a document structure tree to a book. In most cases, `fmbook::` may be more appropriate. Note that: • This axis does not consider element names; therefore, the node test should always be an asterisk (*) • Like `fmbook::`, the axis will match the component element regardless of the current context in the document. That is, the context does not need to be the document HLE.

Axis	Behavior
`fmchap`	Matches the HLE of the document associated with the current book component element; that is, the corresponding chapter file. It only matches if: • The current context is a component-level element within a book structure tree • The associated chapter is currently open, unless the expression is being used by a feature that also supports automatic file opening, such as Node Wizard scripts For example, assuming that a book is active, the following expression will match the HLEs of all chapter documents of the book: **`//*/fmchap::*`** If no book is active, the expression would match nothing. Note the following: • This axis does not consider element names; therefore, the node test should always be an asterisk (*). • This axis matches HLEs in the main flow only. You cannot step from a book into any flow other than the main flow.
`fmfile`	Matches the HLE of the file specified as the node test. You can specify: • An absolute path • A relative path (relative the currently-active file) • The filename of any open file, regardless of its actual location in the file system For example, the following expression matches the HLE of the file `somefile.fm`: **`fmfile::somefile.fm`** Note the following: • The axis is valid for both document and book files. For document files, the axis will match the HLE of the main flow only, not any other flow. • If path separators are required, use forward slashes, for example: **`fmfile::C:/MyDocs/somefile.fm`** • If the path contains any whitespace, you must enclose it in quotes, for example: **`fmfile::"C:/My Docs/some file.fm"`** • The target file must be currently open, unless the feature using the expression provides file-opening capabilities, such as Node Wizard scripts;

The following example matches all `Body` elements in an entire book, regardless of whether the book or a chapter file is currently active. It includes a diagram of how the axes are working. For more examples, see *“Cross-book and cross-file queries”* on page 31.

`fmbook` – Matches the book HLE.

`fmchap` – For any elements matched by the previous axis that happen to be component-level elements, matches the HLE of the corresponding chapter (document) file.

```
fmbook::*//*/fmchap::*//Body
```

Starting from the context of the book HLE, matches all descendant-or-self elements. That is, all elements in the book structure tree.

Starting from the context of the document HLE, matches all descendent-or-self elements that are **`Body`** elements. The behavior is identical to a scenario where you had the document HLE selected, then issued:

`descendant-or-self::Body`

Abbreviated axes

FrameSLT supports most XPath abbreviations for supported axes and functions, as shown in the following examples. If not shown, the abbreviation is not supported.

Abbreviated syntax	Equivalent long version
`/Section/Para`	`/child::Section/child::Para`
`/Section[@Output = "PDF"]`	`Section[attribute::Output = "PDF"]`
`//Section/Para`	`/descendant-or-self::Section/child::Para`
`.`	`self::node()`
`..`	`parent::node()`
`/Section[5]`	`/child::Section[position() = 5]`
`/Body[last()]`	`child::Body[position() = last()]`

Supported logical test operators

Operator	Meaning
= or ==	equals
`!=`	does not equal
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Examples:

`child::Para[position() >= 5]` Select all `Para` children in the fifth position or higher.

`Heading[. != "This is a heading"]` Select all `Heading` children that *do not* contain the text “This is a heading.”

`Conditional[@Output = "PDF"]` Select all `Conditional` children that have an `Output` attribute, and at least one of the values is PDF.

Supported functions

FrameSLT XPath supports the following functions:

- `position()`
- `last()`
- `contains()`
- `contains-ci()`
- `starts-with()`
- `starts-with-ci()`
- `not()`

The following sections describe these functions in more detail.

Node position functions

FrameSLT supports the following position-related functions:

- **position()** Returns an element node's position in a branch relative to its siblings. The behavior of this function differs according to the most recent previous axis. See the W3C documentation for more information.
- **last()** Returns the position of the last element node in the branch containing the context element node

For example, a test for `position() = 3` would only match if the element were in the third position. Or, a test for `last() = 3` would only match if the element were on last on a branch and in the third position.

In an expression, the order of functions and operational terms is unimportant. For example, `position() = 3` means the same as `3 = position()`.

For more detailed examples, see *"FrameSLT XPath examples"* on page 28.

Node content functions

FrameSLT supports the following content-related functions:

- **contains(x,y)** Returns the string "true" if the string "x" contains the string "y", otherwise returns the string "false". This function *is* case-sensitive.
- **contains-ci(x,y)** Returns the string "true" if the string "x" contains the string "y", otherwise returns the string "false". This function is *not* case-sensitive.
- **starts-with(x,y)** Returns the string "true" if the string "x" starts with the string "y", otherwise returns the string "false". This function *is* case-sensitive.
- **starts-with-ci(x,y)** Returns the string "true" if the string "x" starts with the string "y", otherwise returns the string "false". This function is *not* case-sensitive.

Note: `contains-ci()` and `starts-with-ci()` are not part of the W3C XPath recommendation. They are "add-on" functions provided with FrameSLT for your convenience.

All of these functions require two arguments, which can either be a literal string or a node test. In the case of a node test, the content of the matched node becomes the string for comparison when the function is evaluated. If any node test for any argument fails, the function will return "false."

As an example, the following function will return "true" if a child `Heading` element contains the text "mytext":

```
contains(Heading,"mytext")
```

The following function will return “true” if the current context node contains this text:

```
contains(.,"mytext")
```

Functions such as these are used in predicates, and if a string comparison operator is missing, the parser assumes a match of “true” to satisfy the predicate. Therefore, the following XPath expressions are functionally equivalent:

```
//*[contains(., "mytext")]
//*[contains(., "mytext") = "true"]
//*[contains(., "mytext") != "false"]
```

These expressions will all match any element in the tree that contains the text string “mytext”.

For more detailed examples, see *“FrameSLT XPath examples”* on page 28.

Boolean functions

FrameSLT supports the following Boolean-related function:

not() Returns either the string “true” or “false”, intending to represent the opposite of the return of its argument.

`not()` always takes a single argument. If the argument is a node test (that is, returns a node value), the function will return “false” if a node is found, otherwise it returns “true”. For example, the following function will return “true” only if a child `Heading` element does not exist, with respect to the current context:

```
not(Heading)
```

Or as another example, the following function will return true only if the context element itself is not named `Heading`:

```
not(self::Heading)
```

If the argument returns a string value, `not()` will return “true” only if the return string is empty or equals “false”. For example, the following functions will return “true”:

```
not("")
not("false")
```

Besides literal strings, any argument that returns a literal string, such as another function, is evaluated in the same fashion. For example, the following function will return “true” only if the context node does not contain the text “mytext”:

```
not(contains(.,"mytext"))
```

The `not()` function is a powerful tool that can make XPath queries more precise, but the logic can quickly become complex. For more detailed examples, see *“FrameSLT XPath examples”* on page 28.

Node test wildcards

FrameSLT supports the asterisk (*) wildcard for node testing, which indicates “any” element node. For example, the following expression will match every element in the document:

```
//*
```

The asterisk will not match text nodes, and it must appear alone. For example, you cannot use:

```
//B*dy
```

...to match `Body` elements.

EDD-applied prefixes/suffixes and node testing

When you test an element for content, such as in the following expression:

```
//Section[Heading = "My Heading"]
```

...no prefixes or suffixes applied by the EDD are considered. Therefore, in the example above, the `Heading` element would have to contain the text “My Heading” as typed by an author, and any EDD prefixes and/or suffixes are completely ignored.

Unsupported syntax

The following types of XPath syntax are not supported by FrameSLT:

Parenthetical expressions in compound logical tests

Compound logical tests are supported, but not with parenthetical expressions. Therefore, compound conjunctions are also not supported. For example, the following expression cannot be processed:

```
Body[. = "MyText" and (last() or 5)]
```

Because “back-to-back” predicates are considered to have an “and” logic, the following expression is also not supported:

```
Body[. = "MyText"][5 or last()]
```

However, all of these situations can be replicated in a longer form, using the “self” axis and multiple predicates, for example:

```
Body[. = "MyText" and .[last() or 5]]
```

Abbreviated attribute and value test

The following abbreviated syntax for testing an attribute value is not supported:

```
Body[@Output("PDF")]
```

Instead, use the following:

```
Body[@Output = "PDF"]
```

Standalone “go-to-root” XPath expressions

The following expression has no relevance in FrameSLT and is therefore not supported:

```
/
```

With XSLT, you might see this XPath expression frequently in `template` elements, such as `<xsl:template match="/">`. However, this concept has no application in FrameSLT and therefore the expression cannot be parsed.

Direct syntax to unique ID attribute nodes

The following syntax, used to select an element node with a particular unique ID attribute, is not supported:

```
ElementName("ID")
```

For example, the following expression, used to find a child `Body` element with the “MyID” unique ID, cannot be parsed:

```
Body("MyID")
```

If you require a query using a unique ID attribute, use the attribute name directly. For example:

```
Body[@ID = "MyID"]
```

Limitations and known issues

The following sections describe known discrepancies between the established XPath standard and FrameSLT XPath.

Testing element node text

When testing the text of an element node, only the first paragraph is tested. This includes expressions with whole string evaluations and expressions with functions such as:

```
//Section[Heading = "MyHeading"]
//Body[contains(.,"some text")]
//BulletList[starts-with(.,"R")]
```

This limitation is set because test strings could otherwise become enormously long, such as testing the text of the highest-level element of a 200 page document. Strings of this length would adversely affect performance and likely cause crashes. If you need to test the text in a higher-level element, consider using predicates to test subordinate elements, accomplishing the same goal while reducing the processing strain. For example, instead of:

```
//Section[contains(.,"some text")]
```

...you could use an expression such as:

```
//Section[descendant::*[contains(.,"some text")]]
```

or the following equivalent expression:

```
//Section[contains(*,"some text")]
```

This limitation does not apply to testing attribute values. For attribute nodes, all text of all values is always tested.

Finding text() nodes with no siblings

All elements that contain text also have an implied text node, the text itself. While FrameSLT supports the text() node test, it will not find any text nodes that have no siblings. That is, it a text() node has no element node siblings, FrameSLT XPath is currently unable to find it.

It is hoped that this issue should rarely be of importance in FrameSLT functionality, because FrameMaker’s internal representation of structure would make it difficult to support such XPath constructions. For Node Wizard functions, you can use actions such as “Wrap contents in” and “Paste clipboard over contents” to work around the issue.

Comparing two nodes without a bracketed predicate

In most cases, FrameSLT supports the shorthand syntax for testing node content, such as:

```
//Heading = "My Heading"
```

...which is equivalent to:

```
//Heading[. = "My Heading"]
```

The shorter version will not work, however, if you are attempting to compare two node sets. For example, the following expression is not supported:

```
//Heading = Body
```

To accomplish this type of query, you must write it out with an explicit bracketed predicate using a “to self” node, such as:

```
//Heading[. = Body]
```

Normally, these types of comparisons are rare. Note that this limitation applies to the “baseline” expression only. If the test is already within a predicate, the workaround is not necessary. For example, the following expression will work fine:

```
//Section[Heading = Body]
```

In some cases with long, complex expressions, the shorthand format has exhibited problems. In these rare cases, the longer format can be used to work around the bug.

FrameSLT XPath examples

Tips: Always enclose all string literals in single or double quotes. If your literal must contain double quotes itself, enclose the literal in single quotes, and vice-versa.

Do not enclose integers in quotes.

Don't forget the parenthesis on functions, such as `position()`. Without the parenthesis, FrameSLT will think it is simply looking for an element named `position`.

Remember that XPath expressions can become long and complex. Any error, even as small as a single character, will likely cause an expression to fail.

"Single document" queries

The following examples are applicable for querying within a single document; that is, no traversing across books or between separate files.

Expression	Meaning
`Body`	Match all `Body` children of the context node.
`/Body`	Match all `Body` children of the highest-level element (HLE)
`Body[1]`	Match the first `Body` child of the context node.
`//Body`	Match all `Body` descendants of the HLE, and the HLE if it is a `Body`.
`/descendant::Body[1]`	Match all `Body` descendants of the HLE, that are the first `Body` elements in their respective branches.
`Chapter/Section//Body`	Match all `Body` descendants of the `Section` children of `Chapter`
`Chapter/Section//text()`	Match all text node descendants of the `Section` children of `Chapter`
`Body/parent::Section`	Match all `Body` elements with a Section parent
`Body/ancestor::Section`	Match all `Body` elements with a `Section` parent or at least one `Section` ancestor
`Body/ancestor::Section/ancestor::Section`	Match all `Body` elements with at least two `Section` ancestors
`../Body`	Match all `Body` siblings of the context node
`/*`	Match the highest-level element.
`//node()`	Match every element and text node in the tree.
`//*`	Match every element node in the tree.
`//text()`	Match every text node in the tree.
`//*[@Output]`	Match every element node in the tree with an `Output` attribute

Expression	Meaning
`//*[@Output = "PDF"]`	Match every element node in the tree with an `Output` attribute set to PDF. In FrameSLT, if the attribute has multiple values, they are all considered.
`//*[@Output != "PDF" or @Output != ""]`	Match every element node in the tree with an `Output` attribute not set to “PDF” (any of the attribute’s values), or not empty.
`//*[@*]`	Match every element node in the tree that has at least one attribute, regardless of the attribute contents, if any.
`//Body[../@Output = "PDF"]` or `//Body[..[@Output = "PDF"]]`	Match every `Body` element in the tree whose parent has an `Output` attribute set to “PDF”.
`//Body[parent::Section/@Output = "PDF"]` or `//Body[parent::Section[@Output = "PDF"]]`	Match every `Body` element in the tree with a `Section` parent, whose `Output` attribute is set to “PDF”.
`//Body[parent::Section[3]]`	Match every `Body` element that has a `Section` parent, which is third `Section` element on the branch.
`//Body[last() = 5]`	Match every `Body` element in the tree that has exactly four `Body` element siblings.
`//*[5 and 4]`	Matches nothing. An element cannot occupy two positions.
`//*[@Output = "PDF"][5]`	Matches the same thing as: `//*[@Output = PDF and 5]`
`//Section[Heading = "This text"]`	Match every `Section` element node in the tree with a `Heading` child, with the text “This text”.
`//*[position() > 3 or 5 > position()]`	Matches the same thing as: `//*[4]`
`//Heading[. = "This text"]` or `//Heading = "This text"`	Match every `Heading` element node with the text “This text.”
`//Heading[. > "MyHeading"]`	Match every `Heading` element node with text alphabetically greater than “MyHeading”, such as a `Heading` with the text “YourHeading.” ***Note:*** This type of test is more appropriate for text strings with no spaces. If you attempt to alphabetically compare strings with multiple words, the results may not be as reliable.

Expression	Meaning
`//*[@Output = "PDF" or Body = "text" or 5 or 4 or last() or .]`	Match every element node in the tree. The final “to self” (.) test satisfies everything and negates all other logical tests if they fail.
`//*[contains(.,"mytext")]`	Match every element node in the tree that contains the text “mytext”.
`//*[contains-ci(.,"mytext")]`	Match every element node in the tree that contains the text “mytext”, without regard for case-sensitivity.
`//*[contains(@*,"MyValue")]`	Match every element node in the tree that has an attribute that contains the text “MyValue”.
`//*[not(contains(@*,"MyValue"))]`	Match every element node in the tree that does not have any attribute that contains the text “MyValue”.
`//Section[not(Body)]`	Match every `Section` element in the tree that does not have a child element named `Body`.
`//Section[not(Body[contains(&*, "MyValue")])]`	Match every `Section` element in the tree that does not have a child element named `Body` with any attribute containing the text “MyValue”.
`//*[not(self::*[position() = last()])]`	Match every element node that is not the last element in its respective branch.
`//Heading[starts-with(.,"R")]`	Match every `Heading` element that starts with the letter “R”.
`//Heading[starts-with(.,"R") or contains(.,"My Heading")]`	Match every `Heading` element that starts with the letter “R” or contains the text “My Heading”.
`//Section[Heading = Body]`	Match every `Section` element that has a `Heading` child and a `Body` child that both contain exactly the same text.
`//Section[contains(Body,Heading)]`	Match every `Section` element that has any `Body` child that contains the whole text string wrapped in any `Heading` child.

Cross-book and cross-file queries

The following examples use the special axes for traversing between files and books, as described under *“Special book- and file-related axes”* on page 20.

Expression	Behavior
`fmbook::*` -or- **`/fmbook::*`**	**If the context is an active document:** Matches the highest-level element (HLE) of the “parent” book for the active document; that is, the first active book that can be found that contains the active document as a chapter. If no applicable book can be found, it matches nothing. **If the context is an active book:** Matches the HLE of the book.
`//fmbook::*`	Syntax error, because it effectively represents two axes back-to-back (“descendent-or-self” and “fmbook”.
`fmbook::*/*`	**If the context is an active document:** Matches all child elements of the “parent” book HLE. In a traditional book without folders and groups, it would match all component elements. If no applicable book can be found, it matches nothing. **If the context is an active book:** Matches all children of the book HLE.
`fmbook::*//*/fmchap::*`	**If the context is an active document:** Matches the HLEs of all chapter documents in the parent book. **If the context is an active book:** Matches the HLEs of all chapter documents in the book.
`fmbook::*//*/fmchap::*[self::Chapter]`	**If the context is an active document:** Matches the HLEs of all chapter documents in the parent book that have the tag `Chapter`. **If the context is an active book:** Matches the HLEs of all chapter documents in the book that have the tag `Chapter`.
`fmbook::*//*/fmchap::*//Body`	**If the context is an active document:** Matches all `Body` elements in the parent book. **If the context is an active book:** Matches all `Body` elements in the book.

Expression	Behavior
`fmcomp::*/ancestor::*//*/fmchap::*//Body`	**If the context is an active document:** Matches all `Body` elements in the parent book. **If the context is an active book:** Matches nothing. `fmcomp` is only relevant when the context is a document.
`fmfile::somefile.fm`	**In all contexts:** Matches the HLE of a file with the name `somefile.fm`. The remainder of the absolute path is not relevant, unless you are attempting to open the file, in which case the file must be in the same folder as the current context file.
`fmfile::somefile.fm//Body`	**In all contexts:** Matches all `Body` elements in the file `somefile.fm`.
`fmfile::somefile.fm/fmfile::someotherfile.fm/Body`	**In all contexts:** Matches all `Body` elements in the file `someotherfile.fm`, provided that an HLE for `somefile.fm` was found first. The expression effectively steps through multiple documents in a single query.
`fmfile::somefile.fm/fmbook::*//*/fmchap::*//Body`	**In all contexts:** Matches all `Body` elements in the parent book for the file `somefile.fm`.

Flow-related queries

The following examples use the special axes for traversing between flows, as described under *“Special flow-related axes”* on page 17.

Expression	Behavior
`flow-body::A`	Matches the first text frame of each flow A on the body pages of the document.
`flow-body::*`	Matches nothing, unless there is a flow on the body pages named “*”. Wildcards are not supported for flow-related axes.
`flow-any::A`	Matches the first text frame of each flow A in the document, regardless of page type.
`flow-body::A/*`	Matches each HLE in each flow A on the body pages of the document.
`flow-body::A/Chapter`	Matches each HLE named `Chapter` in each flow A on the body pages of the document. For any flow whose HLE is not named `Chapter`, no match is made.

Expression	Behavior
`flow-body::A//Table`	Matches every `Table` element anywhere within any flow A on the body pages of the document.
`flow-body::A//Table/flow-body::B/*`	Matches each HLE in each flow B on the body pages of the document, provided that there is at least one `Table` element somewhere within any flow A.

Chapter 3 The Node Wizard and Other Utilities

This section contains information on FrameSLT utilities including:

- *“The Node Wizard”* on page 35
- *“Node Wizard scripts”* on page 45
- *“FrameSLT condition management”* on page 69

The Node Wizard

The Node Wizard is an XPath-based search tool that you can use to perform highly-customized queries on your structured documents, and if desired, perform structure manipulation such as element wrapping and setting attributes. Because it is XPath-based, your ability to find specific nodes is extremely versatile and limited only by the extent of the markup available for evaluations.

In some respects, the Node Wizard resembles a traditional “Find/Replace” tool, in which you specify a search criteria and perhaps an optional action to take when the item is found. Unlike FrameMaker’s native Find tool, however, the Node Wizard can use XPath to evaluate nearly any markup quality of an element or attribute during its queries, and perform a host of useful actions when it finds its targets.

Node Wizard searching

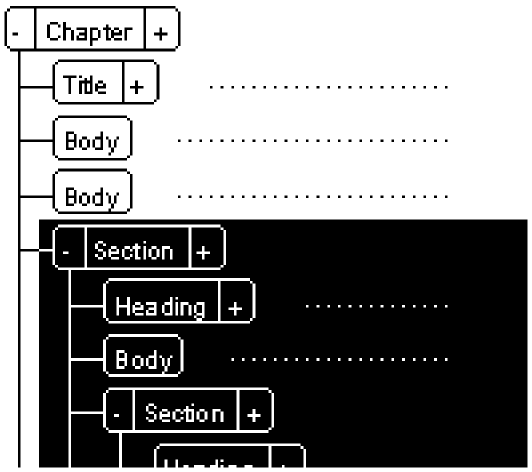
You can use the node wizard as a “search-and-act” tool, or simply as a search tool. In either case, the internal search methodology and XPath handling is the same. For the most part, if you understand XPath, searching with the Node Wizard is intuitive and requires little explanation. However, the Node Wizard does have certain characteristics which you should understand before using it, as explained in the following sections.

Match First, Match Next, and context nodes

An XPath query is a context-based process, during which you begin at a certain point, and each successive query is dependent on the context of the previous query. Any given query has a definitive starting and ending point, unlike a general search, which can be circular. Hence, the Wizard requires both **Match First** and **Match Next** buttons. **Match First** starts (or restarts) the search at the appropriate context node, and **Match Next** resumes the query from the context of the previous query.

As such, an awareness of the original context node is important. If your XPath expression begins with a go-to-root axis (“/”), the process is simple. The original context node is set at the structural root and you do not need to be concerned with it. However, if your XPath does not begin with this axis, your original context becomes the currently selected element node when you click **Match First**. If no node is completely selected, the context node becomes the element that is the direct parent of the insertion point. Therefore, if you are using XPath that does not begin with a slash, you must remain conscious of where you have set the starting context.

As an example, consider the following structure tree, with a `Section` element selected:



If you use the following XPath expression:

`//Body`

...your first query will find the first `Body` element in the tree, sibling to the `Title` element. However, if you use the following XPath expression:

`Body`

...your query will find the first `Body` element under the selected `Section` element, because in the absence of any other context, the `Section` element becomes the original context. After the query, if you clicked **Match First** again, it would find nothing, because the newly-selected `Body` element would be set as the original context and this element has no children at all.

For simplicity, you should use XPath that begins with the “go-to-root” axis whenever possible. If you are performing document- or book-wide node actions with the **Perform Actions On All** button, this axis is required.

Additionally, note the following:

- Structural changes, such as deleting and wrapping elements, can potentially destroy the original contexts and cause subsequent **Match Next** actions to fail. For this reason, the **Match Next** button may become disabled after performing one of these actions and require the query to be restarted with **Match First**. Note that during a **Perform Actions on All** operation, all possible XPath node matches are found and stored before any actions are performed, preventing the need to mimic this behavior during automated actions. For more information about FrameSLT behavior during **Perform Actions On All**, see *“Query behavior during “Perform Actions On All” operations”* on page 44.
- In your FrameSLT preferences, you can specify whether or not the focus should automatically return to the active document after clicking **Match First** or **Match Next**. For more information, see *“Preferences”* on page 10.

Match All

The **Match All** button finds every match according to current Node Wizard settings and places them in the match history, after which you can use the **<<** and **>>** buttons to shuffle through them. The button has the same effect as if you clicked **Match First** once, then clicked **Match Next** repeatedly until all matches were exhausted.

You can set your preferences to optionally report on the number of matches each time you click the button. For more information, see *“Preferences”* on page 10.

XPath favorites

In the **XPath Favorites** area, you can manage a simple list of frequently-used expressions or frequently-used prefixes. For example, you may want to store the following in your favorites:

```
fmbook::*//*/fmchap::*
```

...which is the standard beginning of an expression for performing a book-wide query. Note that you can also manage the list directly in the text file where it is stored. This file is named `FrameSLT_XPath_Favorites.txt` and resides wherever your other settings are stored.

About the Node Wizard and document flows

The Node Wizard can be used to query any structured flow in a document. In addition, you can have Node Wizard actions implemented in any structured flow. The consideration of which flow to process depends on the current location of the insertion point and the status of the **Process/query all structured flows** checkbox, as follows:

Process/query all structured flows is **unchecked**	• For document operations, all processing will occur within the flow that contains the current insertion point. If there is no insertion point, FrameSLT will assume the main flow. • For book operations, FrameSLT will always assume the main flow. For example, if you are processing a document and you click **Match First**, the query will begin in the flow that contains the current insertion point. If you did not have any insertion point established, the query will begin in the main flow, usually flow “A.” The query will not expand to any other flow, unless you reset the insertion point to another flow and click **Match First** again. **Perform Actions On All** exhibits similar behavior. If you have an insertion point established in a flow, the action(s) will be performed in that flow. If there is no insertion point, the action(s) will be performed in the main flow. If you are processing a book, the action(s) will be performed in the main flow of all documents in the book.
Process/query all structured flows is **checked**	For a book or document operations, processing will occur in all structured flows, starting from the first one FrameSLT finds in the document(s). The current location of the insertion point is ignored. For example, if you are processing a document and you click **Match First**, FrameSLT will start the query in the first structured flow it finds. It will continue the query through all structured flows in the document until it finds a match. If it gets through all structured flows without making a match or exhausts all matches, it will then report “Not found.” If you are searching a whole book, the same process will occur for all documents in the book, except that it will not report “Not found” until every document in the book has been queried. A similar behavior occurs for **Perform Actions On All**. If you are processing a document, the action(s) will be performed on all flows in the document. If you are processing a book, the action(s) will be performed in all structured flows in all documents in the book.

About the Node Wizard and file opening

Using the **Allow Node Wizard to open files** option, you can control whether the Node Wizard is allowed to open closed files. It is only applicable to the use of special axes that allow queries to traverse files and books (`fmbook::`, etc). The following table describes the various behaviors, according to axis. In all cases, if the option is not checked and the requested file is not open, the query fails at that point. If the option is checked:

Axis	Behavior
`fmfile`	The Node Wizard attempts to open the specified file. If it cannot be opened, the query fails at that point.
`fmbook`	The query fails at that point. The Node Wizard cannot attempt to open an unknown book. Therefore, the option effectively has no effect on this axis.
`fmcomp`	The query fails at that point. The Node Wizard cannot attempt to open an unknown book. Therefore, the option effectively has no effect on this axis.
`fmchap`	The query attempts to open the respective chapter file. If it cannot open the file, the query fails at that point.

Changing documents/elements during a query

When you click **Match First** or **Match Next**, FrameSLT stores all applicable contexts in memory, and resumes from those contexts the next time you click **Match Next**. Therefore, manually changing the active document and/or the current element selection after a query will not affect the query sequence if you click **Match Next** again, nor will it obstruct the functionality of the match history features (see *“Match history”* on page 38). That is, the query will always resume from the context of the previous match.

If you use special axes that allow queries to traverse files and books (`fmbook::`, etc.), the Node Wizard will automatically manage the movement between files, as applicable. Because XPath can move between files, the Node Wizard displays two different file labels below the **XPath** box:

- **Currently active file** - The file that was active during the last refresh of the Node Wizard. This file may have been activated by a query or by a manual user action.
- **Original query file** - The file that was active the last time that you clicked **Match First**, **Match All**, or **Perform Actions(s) On All**; that is, the original context of the query. This file does not change with a **Match Next** action because the original context does not change.

For more information on axes that traverse files, see *“Special book- and file-related axes”* on page 20.

Match history

The Node Wizard includes **<<** and **>>** buttons that allow you to shuffle through the history of element nodes matched by the XPath expression, from the use of the **Match First**, **Match Next**, and **Match All** buttons. When you step through the history, you are stepping through the history of XPath queries only. No XPath processing takes place when you use these buttons; however, structural alterations that occurred since the history was stored may interfere with their ability to find the original elements. When moving through the history, if the match number is preceded by an exclamation point (!), the original element could not be found again.

Tip: The XPath match history is reset with each successful query initiated by the **Match First** button.

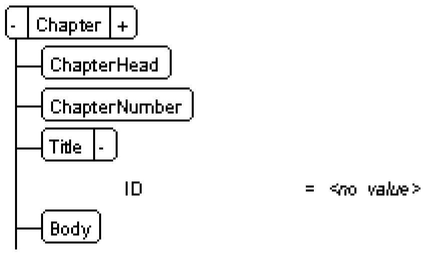
Attribute nodes

In your XPath, you can specify a query for an attribute node. You should be aware, though, of how attribute node matching behaves with regard to Node Wizard functions, as follows:

- **Node selection, after a “match” action** When you click **Match First** or **Match Next**, FrameSLT selects an element in the document if a match is made. If the XPath matches an element node, it will select that element. If it matches an attribute node, it will select the element that contains the matched attribute. In the case of an attribute node, you will not know from the element selection alone which attribute was matched; you will only know the element that contains the matched attribute.

- **Element actions** When the XPath matches an element node, the specified element action occurs on the matched element. When the XPath matches an attribute node, the element action occurs on the element that contains the matched attribute node.
- **Attribute actions** Attribute actions may occur on a matched attribute node, a specified attribute node, or both, depending on the XPath setup and action type. This situation can become complex and is explained in more detail under *"Attribute actions"* on page 41.

As an example, consider the following structure tree:



With this structure, the following two expressions would produce the same result when "Match First" is clicked. That is, both would select the `Title` element:

`/*/@ID`

`/*[@ID]`

Performing node actions

As an option, you can perform a variety of element and attribute actions during your queries. The following sections describe the behavior of these actions in more detail.

Element actions

Note: Unless specifically indicated in the respective description, you should assume that an action works on element nodes only, not text frames matched with a special flow axis (see *"Special flow-related axes"* on page 17).

- **Retag as** Retags matched element nodes with the selected tag name. That is, it changes the element name.
- **Wrap element in** Wraps the matched element node and all its contents in a new element with the selected tag name.
- **Wrap contents in** If the XPath matches an element node, wraps the entire contents of the matched element node in a new element of the selected tag. The new element becomes the first and only child of the original matched element.

  -or-

  If the XPath matches a text frame using a special flow axis, wraps the entire contents of the respective flow in the specified element. This action works on both structured and unstructured flows, which in the case of the latter, wraps all previously-unstructured content in a new HLE.
- **Unwrap** Unwraps (and discards) the matched element node tag. The contents of the element are preserved and placed on the structure tree where the original element resided.
- **Move up** Slides the matched element node up its branch one position, making it the previous sibling of its formerly previous sibling. If the element is already at the top of its branch (that is, the first child of its parent), this action has no effect.

- **Move down** Slides the matched element node down its branch one position, making it the following sibling of its formerly following sibling. If the element is already at the bottom of its branch (that is, the last child of its parent), this action has no effect.
- **Promote** When an element is promoted, it becomes a sibling of its former parent. After promotion, it appears immediately after its former parent. The siblings that follow it become its children.
- **Demote** When demoted, an element becomes a child of the sibling element before it.
- **Delete element** Deletes the matched element node and all its children.
- **Delete contents** If the XPath matches an element node, deletes the entire contents of the matched element node, leaving an empty element.

  -or-

  If the XPath matches a text frame using a special flow axis, deletes the entire contents of the respective flow.
- **Insert elem before** Inserts a new, empty element of the specified tag directly before the matched element node, as its immediate previous sibling.
- **Insert elem after** Inserts a new, empty element of the specified tag directly after the matched element node, as its immediate following sibling.
- **Insert first child** Inserts a new, empty element of the specified tag as the first child of the matched element node.
- **Insert last child** Inserts a new, empty element of the specified tag as the last child of the matched element node.
- **Assign conditions** Assigns the specified conditions to the matched element node and its children. When adding condition tags to assign, the drop-down list is populated based on condition tags found in the currently-active document. However, you may specify any tag. If the Node Wizard attempts to assign a tag that does not exist in the current document, the error report will indicate as such. For important notes on conditional text, see *“A word on conditional text”* on page 45.
- **Paste CB over elem** Pastes the current contents of the Windows clipboard over the matched element node, replacing the element. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.
- **Paste CB over contents** Pastes the current contents of the Windows clipboard over the contents of the matched element node, replacing the original contents but preserving the element tag. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.
- **Paste CB at beginning** Pastes the current contents of the Windows clipboard at the beginning of the matched element node, as the first child. It does not replace any existing content of the matched element. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.
- **Paste CB at end** Pastes the current contents of the Windows clipboard at the end of the matched element node, as the last child. It does not replace any existing content of the matched element. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.
- **Paste CB before** Pastes the current contents of the Windows clipboard directly before the matched element node, as its immediate previous sibling. It does not replace any existing content of the matched element. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.

- **Paste CB after** Pastes the current contents of the Windows clipboard directly after the matched element node, as its immediate following sibling. It does not replace any existing content of the matched element. Clipboard contents may include text, FrameMaker objects, and structural elements. You must copy the desired contents onto the clipboard before running this action.

Attribute actions

As implied by the name, attribute actions involve attribute nodes. The variety and flexibility of FrameSLT attribute manipulation, however, can make it difficult to understand the more advanced capabilities of the plugin. This section attempts to describe the fundamentals of attribute actions in detail, giving you the basic knowledge to understand the bigger picture and possibilities for attribute manipulation. Please read this entire section before using attribute actions in the Node Wizard.

Every attribute action operates on one or more attribute nodes. The particular attributes on which actions operate fall into two categories:

- **Specified attributes** In the Node Wizard, you can directly specify attribute names on which you want the action to occur. With specified attributes, it is not necessary for the XPath expression to match attributes, although it can if you desire. When actions are performed, for each XPath match, FrameSLT will simply look for the specified attribute(s) on each matched element, and generate a warning if the attribute is not found. If the XPath matches attribute nodes, the same behavior will result, with FrameSLT searching the parent element for the specified attribute.

  In short, when an attribute action is performed on a specified attribute, any matched attribute is generally not relevant, other than its contribution to the XPath query. The attribute is acted upon based on the name specified in the Node Wizard.
- **Matched attributes** As an alternative to directly specifying attributes, attribute actions can be performed on attributes matched by the XPath expression. Acting upon matched attributes allows you to be much more precise about attribute manipulation, because the XPath expression will control which attributes are acted upon. Furthermore, it allows you the flexibility to use XPath wildcards to find attribute candidates for the specified action. To indicate that an attribute action should be performed on matched, versus specified, attributes, you should put the text `{xpath-match}` in the Attributes box or simply leave it empty.

Tip: The *Node Wizard Tutorial* explores this subject in more detail and provides hands-on examples. If these explanations do not make sense, the tutorial may help you understand them better.

Note the following important items regarding specified versus matched attributes:

- You may combine matched and specified attributes in a single action. For example, if both “Product” and “{xpath-match}” are listed in the Attributes box, the specified action will occur on both the `Product` and the matched attribute node for each match, as applicable.
- If you specify an attribute that is not found on the matched or nearest element node, an error will occur, on an action-by-action basis.
- If you specify actions to occur on matched attributes, but your XPath does not match attribute nodes, the attribute action will have no effect and will generally be considered an error.

Furthermore, certain attribute actions inherently apply to both matched and specified attributes, and therefore require both a specified attribute and an XPath expression that matches attribute nodes. The following list describes the available attribute actions in detail, noting specified versus matched attribute issues where appropriate.

- **Add specified values** Adds the specified values to the specified attributes in addition to any existing values. Specified attributes may include “{xpath-match},” which causes the specified values to be set on any attribute matched by the XPath expression.

- **Remove specified values** Removes the specified values from the specified attributes, if they currently exists. values. Specified attributes may include "{xpath-match}," which causes the values to be removed from any attribute matched by the XPath expression.
- **Replace values with spec** Replaces any current values of the specified attributes with those specified in the dialog. All current values of these attributes are removed first. Specified attributes may include "{xpath-match}," which causes the specified values to be replaced on any attribute matched by the XPath expression.
- **Delete all values** Clears all current values from the specified attributes, leaving empty attributes. Specified attributes may include "{xpath-match}," which causes the deletion of current values from all attributes matched by the XPath expression.
- **Move value to elem text** For all specified attributes, moves the current attribute value to the parent element text, replacing any element text that previously existed. The element/attribute pair on which this occurs is based on matches by the XPath. For best results, only a single attribute should be specified, and it may be specified as "{xpath-match}," causing text movement from the attributes matched by the XPath expression. As a "move" operation, the value is physically moved and the attribute is left empty. This action operates on the first attribute value only and is limited to 255 characters.
- **Copy value to elem text** For all specified attributes, copies the current attribute value to the parent element text, replacing any element text that previously existed. The element/attribute pair on which this occurs is based on matches by the XPath. For best results, only a single attribute should be specified, and it may be specified as "{xpath-match}," causing a text copy from the attributes matched by the XPath expression. As a "copy" operation, the value is copied only and the original attribute is unaffected. This action operates on the first attribute value only and is limited to 255 characters.
- **Move elem text to value** For the specified attributes, moves the parent element text to the first attribute value, replacing any attribute values that previously existed. Multiple attributes may be specified, although a single attribute specification is generally recommended for management purposes. The element/attribute pair on which this occurs is based on matches by the XPath. The specified attribute may be specified as "{xpath-match}," causing text movement to the attributes matched by the XPath expression. As a "move" operation, the value is physically moved and the element is left empty. This action operates on the first attribute value only and is limited to 255 characters.
- **Copy elem text to value** For the specified attributes, copies the parent element text to the first attribute value, replacing any attribute values that previously existed. Multiple attributes may be specified, although a single attribute specification is generally recommended for management purposes. The element/attribute pair on which this occurs is based on matches by the XPath. The specified attribute may be specified as "{xpath-match}," causing a text copy to the attributes matched by the XPath expression. As a "copy" operation, the value is copied from the element only and the element is left as it was found. This action operates on the first attribute value only and is limited to 255 characters.
- **Move values to spec attr** For each XPath match, moves the values found on the matched attribute to the specified attribute. This action, therefore, requires that the XPath expression to match attribute nodes, not element nodes. Furthermore, the specified attribute should NOT be "{xpath-match}", because the other end of the transaction is already the matched attribute. Only a single attribute should be specified, and any additional attributes will be ignored. As a "move" operation, the value(s) are moved and the matched attribute is left empty. If the matched attribute was empty originally, both attributes will be empty at the end of the operation.
- **Copy values to spec attr** For each XPath match, copies the values found on the matched attribute to the specified attribute. This action, therefore, requires that the XPath expression to match attribute nodes, not element nodes. Furthermore, the specified attribute should NOT be "{xpath-match}", because the other end of the transaction is already the matched attribute.

Only a single attribute should be specified, and any additional attributes will be ignored. As a "copy" operation, the value(s) are copied only and the matched attribute is left as found. If the matched attribute was empty originally, both attributes will be empty at the end of the operation.

- **Move values from spec attr** For each XPath match, moves the values found on the specified attribute to the matched attribute. This action, therefore, requires that the XPath expression to match attribute nodes, not element nodes. Furthermore, the specified attribute should NOT be "{xpath-match}", because the other end of the transaction is already the matched attribute. Only a single attribute should be specified, and any additional attributes will be ignored. As a "move" operation, the value(s) are moved and the specified attribute is left empty. If the specified attribute was empty originally, both attributes will be empty at the end of the operation.
- **Copy values from spec attr** For each XPath match, copies the values found on the specified attribute to the matched attribute. This action, therefore, requires that the XPath expression to match attribute nodes, not element nodes. Furthermore, the specified attribute should NOT be "{xpath-match}", because the other end of the transaction is already the matched attribute. Only a single attribute should be specified, and any additional attributes will be ignored. As a "copy" operation, the value(s) are copied only and the specified attribute is left as found. If the specified attribute was empty originally, both attributes will be empty at the end of the operation.
- **Swap values with spec attr** For each XPath match, swaps the values found on the matched attribute with those found on the specified attribute. That is, the two original value sets are exchanged between the two attributes. This action, therefore, requires that the XPath expression to match attribute nodes, not element nodes. Furthermore, the specified attribute should NOT be "{xpath-match}", because the other end of the transaction is already the matched attribute. Only a single attribute should be specified, and any additional attributes will be ignored.
- **Search and replace string** For each XPath attribute match, performs a string search and replace on existing values, using the specified strings. The search string may represent a whole value or a string fragment within values, and the replace string may be zero or more characters, with an empty replace string simply deleting any instance of the search string. The search string is case-sensitive, and wildcards are currently not supported. The operation is performed on attributes matched by the XPath expression only, and is performed on all existing values of the respective attribute.
- **Remove invalid attribute** For each XPath match, removes the specified attribute(s) from the element if found to be invalid; that is, not defined by the EDD. Removal includes removal of the attribute and all values on the matched element only. Specified attributes may include "{xpath-match}", which will cause the removal of the attributes matched by the XPath expression. If any specified or matched attribute is found to be valid, the action has no effect and will produce a warning as applicable.

IMPORTANT NOTE

The following actions move text from an element to an attribute, or vice-versa:

- Move value to elem text
- Copy value to elem text
- Move elem text to value
- Copy elem text to value

If the original text string to be moved or copied is empty, the action will only proceed if you have your FrameSLT preferences set accordingly. If you choose to allow the moving or copying empty strings, the result will be the deletion of all text at the target. For example, if the action "Copy value to elem text" is performed on an attribute with no values, the result will be the deletion of all current contents of the element, if any. For more information on setting this preference, see *"Preferences"* on page 10.

Important warning about node actions

Node actions, especially when performed as a batch with the **Perform All Actions** button, can cause major changes to your structure and content. Do not use this function unless:

- You are 100% sure that you understand what your XPath and the specified actions are going to do

  and/or
- Your files are securely backed up

Backups are recommended in any case. Remember that you can also close and reopen your files afterwards WITHOUT SAVING CHANGES to restore your files.

If you remember nothing else, remember this: **A SINGLE CLICK OF THE "PERFORM ACTIONS ON ALL" BUTTON COULD DELETE EVERY SINGLE CHARACTER OF CONTENT OUT OF AN ENTIRE BOOK WITHIN SECONDS, IF YOUR PARAMETERS ARE NOT SET UP PROPERLY**. Use the Node Wizard at your own risk and DO NOT SAVE CHANGES unless you are positive that they are what you intended!

"Perform Action(s) and Find Next" button

This button has the effect of clicking **Perform Action(s)** then **Match Next**. Note that some actions, such as element deletion, unwrapping, and promotion/demotion, alter the structure tree significantly enough that the original XPath context is destroyed and the query cannot continue with this button.

Query behavior during "Perform Actions On All" operations

When you click **Perform Actions On All** the Node Wizard will query the active document for all possible nodes that match the specified XPath expression and store them in memory. After the node list is complete, it then steps through the list performing the specified action(s) on each node. This methodology is followed even if the expression matches nodes in multiple files.

For this reason, element actions such as deletion, unwrapping, promotion, and demotion can be reliably performed during **Perform Actions On All** operations, with proper setup. This behavior differs somewhat from using the **Match Next** and **Perform Action(s)** button, because the **Match First** and **Match Next** buttons do not store all possible matches from the outset. Rather, they go one match at a time, each time considering the current context as it exists in the document. Therefore, certain element actions may disable the **Match Next** button when querying a node at a time.

Element actions that preclude attribute actions

The following element actions cannot be combined with an attribute action:

- Unwrap—After unwrapping an element, no attributes are available for an action.
- Delete element—After deleting an element, no attributes are available for an action.
- Paste clipboard contents over element—This action replaces the original element and creates too much uncertainty to safely attempt attribute actions.

Wrapping elements and performing an attribute action

If you combine a "wrap" element action and an attribute action, the attribute action is performed on the element that was originally matched by the XPath expression, not the new, "wrapping" element. For example, if your expression is set to match `Body` elements, and the element action is set to wrap them in `Section` elements, any attribute actions will be performed on the original `Body` elements, not any new `Section` elements.

A word on conditional text

If you have hidden conditional text in your document, it will not be affected by any Node Wizard function. In essence, it is invisible to FrameSLT. If you use conditional text, be sure to manage it carefully when conducting Node Wizard activity.

You can remove all assigned condition tags from your visible text by using the following XPath expression:

```
//*
```

...combined with the “Assign conditions” element action, but with no condition tags specified.

XPath parsing

The Node Wizard includes an option to parse your XPath only and forgo any searching. This function is a convenience to help you check for syntax errors in your XPath. With this option, you can also print the parsed XPath components to the console for rudimentary debugging purposes. This console report can help you see how FrameSLT recognized the components of your XPath and may help you correct errors. For example, if you forget to put the parentheses on a “position()” function, the console report will indicate that the component was recognized as a test for a node named “position,” rather than a logical test involving a node’s position.

FrameSLT parses XPath into a tree-like structure which it navigates through while searching your documents. The console report, therefore, attempts to outline this parsed XPath tree. Please note that the XPath processing is somewhat complex and this console report is not intended to be a comprehensive debugging tool. It is used during the development and testing of FrameSLT and has been simply left there in the event that you might find some use for it as well. With some experience, you should at least be able to see how axes, functions, node tests, and predicates are recognized. For complex expressions and query issues, though, you may need to run experiments until your queries behave as expected.

Node Wizard scripts

Node Wizard scripts allow you to automate sequences of element and/or attribute actions throughout an entire document or book, without the need to configure the Node Wizard dialog each time. A script can perform any action supported by the Node Wizard and more, in any sequence and with any frequency. You may have any number of scripts defined, with any number of events.

With the capacity to nest events with cascading XPath context, you can perform very complex document alterations, including the ability to retrieve content from any attribute or element in any file and move it to any other attribute or element. For more information on nesting events, see *“About subevents”* on page 46.

Node Wizard scripts are launched using the scripts dialog at **FrameSLT > Node Wizard Scripts**. Scripts may also be initiated by FrameScript, FrameAC, or any other API client through the FrameSLT external call interface. For more information, see *“`RunNWScript`”* on page 135.

Scripts can also be set to automatically run after key events such as document opening and EDD imports. For more information on autorunning Node Wizard scripts, see *“Autorun triggers”* on page 53.

Tip: FrameSLT includes a tutorial on Node Wizard scripting. It may be a good place to start for understanding how it works.

Note: **LIKE NODE WIZARD ACTIONS, A SINGLE SCRIPT EVENT COULD DELETE EVERY SINGLE CHARACTER OF CONTENT OUT OF AN ENTIRE BOOK WITHIN SECONDS. USE THIS SCRIPTING AT YOUR OWN RISK. KEEP BACKUPS AND DO NOT SAVE ANY FILES UNTIL YOU ARE SURE THAT A SCRIPT IS DOING WHAT YOU INTEND IT TO. A SINGLE MISTAKEN CHARACTER IN A SCRIPT COULD CAUSE IT TO DO SOMETHING COMPLETELY UNEXPECTED AND POTENTIALLY**

CATASTROPHIC, ESPECIALLY IF YOU SAVE THE RESULTS WITHOUT VERIFYING THEM.

About Node Wizard scripts

A script is a series of one or more XPath-driven events that run in sequence. Each event is much like a single snapshot of the Node Wizard, comprised of an XPath expression and any desired element/attribute actions. When a script runs, each event runs independently and behaves exactly as if you had manually configured the Node Wizard with the respective settings and clicked **Perform Actions On All**. When a script is complete, FrameSLT produces the same status report that you see after a Node Wizard “Perform Actions On All.”

Because events are like snapshots of the Node Wizard, a familiarity with the Node Wizard should be all you need to successfully write and run scripts. For more information on writing and editing scripts, see *“Writing and editing scripts - General information”* on page 50. For more details on the behavior of element and attribute actions themselves, see *“Element actions”* on page 39 and *“Attribute actions”* on page 41.

About subevents

In a Node Wizard script, you can nest events using the `SubEvent` element at the end of any `NWScriptEvent` or `SubEvent` element branch. A subevent is run for each match of the parent event or subevent XPath, with the XPath query launched from the context of that match. There is no limit to the depth of subevent nesting.

To use subevents successfully, it is important to understand the concept of cascading XPath context and the looping aspects of a parent event. As an example, assume that you have a parent event with the following XPath:

```
//Section/@MyAttribute
```

...and a subevent with the following XPath:

```
Body
```

Each time the parent event XPath matches a `MyAttribute` attribute, it will perform its respective actions on that match and then launch the subevent, whose XPath query will start from the element context of that match. In summary, therefore, note the following:

- A subevent is launched once for each match of the parent event XPath, with a new subevent XPath query performed each time.

  Note: If a subevent deletes an element in the list of parent event matches, the script will not launch any subevents for that iteration. This behavior is necessary because the context of a non-existent node can cause unpredictable behavior within a subevent. Therefore, you should be very careful that a subevent doesn’t delete or unwrap any element in the list of parent event matches.
- The XPath query of a subevent starts from the context of the parent event match. For this reason, subevent expressions do not require the “go-to-root” axis (`/`) like top-level events. You can use this axis for subevent expressions, however, understanding that the context passed to the subevent will then be overridden and therefore irrelevant.
- A parent event will perform its own element/attribute action(s), if any, before passing the context to the subevent and launching it.
- If a parent event makes no matches, a subevent will never run.
- Only the element node context is passed to a subevent, even in the case of an attribute match. This is important to allow further contextual queries within subevents when a parent event matches attributes. Technically, the context of an attribute is a dead-end from which no further navigation is possible, so a rigid adherence to this context would limit the ability to move away from an attribute context and perform actions elsewhere. If you need to work on a matched attribute again within a subevent, set up your XPath expression to match it again.

- The context of any given event is passed to subevents only. It is never passed to sibling events.
- Any event can have zero, one, or more subevents, which must be located at the end of its element branch.

Subevent nesting in conjunction with clipboard features provide a very powerful and unique transformation engine for structured FrameMaker documents. You can copy content from any element or attribute in a parent event and paste it anywhere else with subevents. Use it with caution.

About document versus whole-book processing

All scripts begin operation on the currently-active file, whether it is a document or a book. At any time, any event can shift processing to a different file, using the special axes described under *"Special book- and file-related axes"* on page 20.

When a script is run on a book, you have two different methods to direct initial processing:

- **Strict XPath-based navigation (default)** - This method requires explicit XPath syntax to control the navigation among chapter files. That is, the script does not automatically iterate through all chapter files; rather, it goes directly to any location specified by XPath expressions. The initial starting context for any top-level expression is the book structure tree HLE.

  As an example, assume that you would like to match all `Body` elements in all chapters in a book, perhaps to retag as something else. You could use an event XPath expression such as:

  ```
  fmbook::*//*/fmchap::*//Body
  ```

  With this type of expression, you can run the script on the book file and the query will traverse the entire book. Note that by nature of the initial navigation to the book HLE, this example would operate identically if run on a chapter file instead.

- **"Classic" iterative chapter navigation** - This method iterates through all chapters in the book and runs the script independently on each chapter. The initial starting context with each chapter is the HLE of the respective chapter tree. The book structure tree is ignored, unless an XPath expression explicitly directs the navigation from the document tree to the book tree.

  To enable this mode, you must insert a `<BookNavigationMode>` element under the script `<GeneralSettings>`, then a `<Classic>` element under that.

  Note: In classic mode, all book-based XPath syntax is fully supported, so you can still navigate through the book structure tree and into other documents if desired.

Note: Extended XPath axes and related functionality offer an extensive amount of flexibility with XPath queries, but at the cost of increased complexity. An expression can now find nearly anything in nearly any file, but only if you know how to write it. Please contact West Street for assistance if necessary. We are happy to help you get the most out of this software.

Element/attribute actions supported by scripts only

This section describes element and attribute actions that are supported by scripts but not by the Node Wizard dialog box. Certain actions are not useful without the ability to nest events and provide multiple XPath expressions. For descriptions of all other actions that are supported by the dialog box, see *"Element actions"* on page 39 and *"Attribute actions"* on page 41.

Element actions:

- `Copy_elem_to_CB` - Copies the entire matched element to the clipboard, including all contents and descendant elements.
- `Copy_elem_contents_to_CB` - Copies the contents of the matched element to the clipboard, including all descendant elements, but does not copy the element itself.

  Note: "Copy" actions are typically used in conjunction with a subevent that pastes the content somewhere else.

- `Copy_elem_text_to_parameter` - Copies the text content of the matched element to the specified parameter, up to the first 2056 characters.
- `Set_elem_text` - Sets the text content of the element according to the source specified by subelements, overwriting any existing content.

Attribute actions:

- `Copy_all_values_to_CB` - Copies all values of the matched attribute to the clipboard. The XPath must match an attribute for this action to work. If values are copied and later pasted to another attribute, they will look exactly as they did at the original attribute. If multiple values are copied and later pasted as element text, they will be pasted as a space-delimited (tokenized) list.
- `Copy_first_value_to_CB` - Copies the first value of the matched attribute to the clipboard. The XPath must match an attribute for this action to work. This action considers whitespace as an attribute value delimiter, in support of tokenized lists.
- `Detokenize_values` - Separates any whitespace-delimited attribute values into individual strings. Note that individual string values are generally applicable within the FrameMaker environment only and do not translate well to XML.
- `Paste_CB_to_matched_attr` - Pastes the current contents of the clipboard to the contents of the matched attribute. The XPath must match an attribute for this action to work. If the clipboard currently contains text and/or element content, any text will be truncated to 254 characters as applicable and pasted as the first and only attribute value.
- `Paste_CB_at_beg_of_matched_attr` - Prepends the current contents of the clipboard to the contents of the first value of the matched attribute. The XPath must match an attribute for this action to work. This action includes the following contingencies:
    - If the attribute currently contains no values, the action behaves like the `Paste_CB_to_matched_attr` action.
    - If the attribute contains multiple values, the action only acts upon the first value.
    - If the clipboard contains multiple attribute values, the action only prepends the first value.
- `Paste_CB_at_end_of_matched_attr` - Behaves identically to `Paste_CB_at_beg_of_matched_attr`, except that the value is appended, not prepended.
- `Set_attr_value` - Sets the value of the attribute according to the source specified by subelements, overwriting any existing values. This action is somewhat of an extension of `Replace_values_with_spec`; however, note that it can set a single value only.
- `Sort_values` - Allows alphabetical and numeric sorting of values, either ascending or descending. Values are sorted and then set.
- `Tokenize_values` - Converts all values represented as individual strings into a single whitespace-delimited string. This action is useful to convert the FrameMaker convention of individual strings for individual values into a format more compatible with XML.

About the script settings file

The key component of the scripting feature is the “script settings” file, where all scripts are stored and subsequently referenced. This file is named:

`FrameSLT_Node_Wizard_Scripts.fm`

...and resides in the folder where your FrameSLT settings and support files are located, either your “user profile” area or the FrameSLT installation folder. This file is a structured document that contains all the definitions and parameters of your currently active and inactive scripts.

Note the following about this file:

- This file is the only interface for editing scripts. FrameSLT currently has no graphical support for script editing. For more information on working in this file, see *“Writing and editing scripts - General information”* on page 50.

- This file must remain in the settings area with its original name, otherwise FrameSLT will not be able to find it. In the future, we may enhance FrameSLT such that the location of this file is customizable, but this effort will only be undertaken based on user demand. If you have a need for this enhancement and/or more flexibility with script file location, please contact West Street.
- You should not alter the structure rules of this file, because FrameSLT is expecting to find certain elements in certain places. If you have a need to alter the EDD for formatting purposes, please contact West Street first, because we can advise you of what will and will not break the scripting process.

Running Node Wizard scripts within FrameMaker

All scripts and events are run through the scripts dialog, accessible by selecting **FrameSLT > Node Wizard Scripts**. This dialog contains the following features and controls:

Script control	Description
Active scripts box	Lists all currently-active scripts in the scripts settings document. For more information on active vs. inactive scripts, see *“Script-level general settings”* on page 51.
Script events box	Lists all events for the selected script. If the event has a name, the name will be listed. For more information on event names, see *“Event name and description”* on page 54.
Script description and **Event description**	Displays descriptions for the selected script and event, as applicable. If a script or event has no description, the respective box will be grayed out. For more information on script and event descriptions, see *“Script name and description”* on page 50 and *“Event name and description”* on page 54.
Run Script	Runs the selected script on the currently-active document or book. ***Tip:*** If you have the script settings file open for editing, be careful not to accidentally run the script on the settings file itself.
Check Script	Runs basic error checking on the selected script, identical to the error checking that occurs when you run a script. This button does not run the script.
Edit Script	For the selected script, jumps to the associated element in the script settings file. If the scripts settings file is closed, you will be prompted to open it.
Run Event	Runs the selected event, independently of any other events. With respect to the parameters in the script settings file, this button is similar to the **Perform Actions On All** button on the Node Wizard. ***Tip:*** If you have the script settings file open for editing, be careful not to accidentally run the event on the settings file itself.

Script control	Description
Check Event	Runs basic error checking on the selected event. This button does not run the event.
Edit Event	For the selected event, jumps to the associated element in the script settings file. If the scripts settings file is closed, you will be prompted to open it.

Note: During each script/event run, FrameSLT must walk through the script settings file to get the parameters. If the file is currently open, these activities will cause the file to think that it has unsaved changes, even if you have not made changes yourself. Therefore, you may be prompted to save changes when you close the file, even if you didn't make any.

Writing and editing scripts - General information

All script writing and editing occurs in the script settings file. For more information on file naming and its location, see *"About the script settings file"* on page 48.

The script settings file is a structured FrameMaker document and must be edited within FrameMaker. It uses structural markup to define scripts, similar to how an EDD uses its own element names to provide structure rule data. For these reasons, you do not need to learn any kind of scripting language. For the most part, you need only to open the document and follow the guidance of the element catalog.

The scripts document must have a highest-level element of `WS_Scripts`, with a child `NodeWizardScripts` and each script wrapped in a `NodeWizardScript` element. No other elements should appear at these levels.

The remaining information in this document describe the technical details of scripts that may not be apparent from the markup alone. You may also find it valuable to study the tutorial and sample scripts that installed with FrameSLT.

Note: The scripts file EDD is intended to guide you through the script writing process and prevent errors. If your settings file is valid against its EDD, you are at least guaranteed that all required elements are in place and that none are functionally missing. For this reason, FrameSLT performs a limited validity check before running a script.

About parameters

Similar to the corresponding XSLT concept, Node Wizard Scripts support the use of parameters, which are effectively variables that you can manipulate and retrieve at will. Note the following general information about parameters:

- The general limit on parameter name length is 255 characters. The general limit on value length is 2056 characters. There is no limit on the number of parameters you may define.
- Parameter usage is supported within XPath expressions (see *"Using parameters in XPath expressions"* on page 55).
- Where appropriate, FrameSLT will automatically attempt to interpret parameter values as integers; for example, within mathematical expressions and as arguments within XPath `position()` function node tests.
- With the exception of parameters set with external calls (see `SetScriptParm` on page 140), all previous parameter data is cleared each time a script is launched.

Script name and description

The first child of a script element must be a `NWScriptName` element, whose text contents represent the name of the script. All scripts must have a unique name. Following the name, an optional description may be wrapped in a `NWScriptDescription` element. If no description element is provided, the “description” box in the scripts dialog will be grayed out when the script is selected in the scripts dialog.

Script-level general settings

A `NodeWizardScript` element must contain a `NWGeneralSettings` element, which may include the following subelements:

Note: For “yes/no” Boolean options, the default is always “no.”

Element name	**Required/ optional**	**Description**
`ScriptIsActive`	Required	Indicates whether the script is active or not by the presence of a `Yes` or `No` subelement. Inactive scripts do not appear in the scripts dialog and cannot be run. The active/inactive mechanism is intended to allow the storage of draft/unused script data in the scripts setting file. ***Note:*** The `Yes` element must be present to activate the script. Otherwise, the script will be assumed inactive.
`ReportElemActionErrors` **`ReportAttrActionErrors`** **`ReportOtherActionErrors`**	Optional	Indicates whether to report errors and warnings associated with element, attribute, and “other” actions, respectively. For element and attribute actions, these options are functionally identical to the “Report errors” checkboxes in Node Wizard. “Other” actions are not supported by the Node Wizard, so no comparable option exists. For each of these options, there must be a `Yes` subelement to enable the option.
`AllowScriptToOpenFiles`	Optional	Indicates whether the script is permitted to open files that are indicated with special axes such as `fmfile::` and `fmcomp::`. This option is functionally identical to the corresponding Node Wizard option (see *“About the Node Wizard and file opening”* on page 37). By default, no file opening is permitted. If an XPath expression points to a closed file, the query will fail at that point.

Element name	Required/ optional	Description
`AssumeTokenizedAttrVals`	Optional	Indicates whether the script should assume that multiple values for attributes are specified as whitespace-delimited strings, in the same convention as XML. If set to Yes, the script will recognize individual values within a tokenized string for the following actions: • `Add_specified_values` • `Remove_specified_values` • `Sort_values` Otherwise, the setting generally has no effect on other attribute actions. Note the following: • **This setting should match the convention you use to specify attribute values.** If it does not, certain actions may convert the convention automatically. For example, if you have the setting enabled and you run an “add values” operation, it will automatically tokenize any values that it sets. • The default is No; that is, do not recognize whitespace as a value delimiter. If you want to convert from string values to tokenized values or vice-versa, use the `Tokenize_values` or `Detokenize_values` attribute action, respectively.
`EnableParamtersInXPath`	Optional	Indicates whether the script is permitted to resolve parameters in XPath expressions. If it is not, parameters will be handled literally as written. This setting is required for parameter usage. For example, if parameters are enabled and the parameter “`parm1`” equals “Body”, the following expression: **`//$parm1`** ...will resolve as: **`//Body`** Otherwise, the query will look for elements with the actual tag “`$parm1`”. For more information on parameters, see *“Using parameters in XPath expressions”* on page 55.

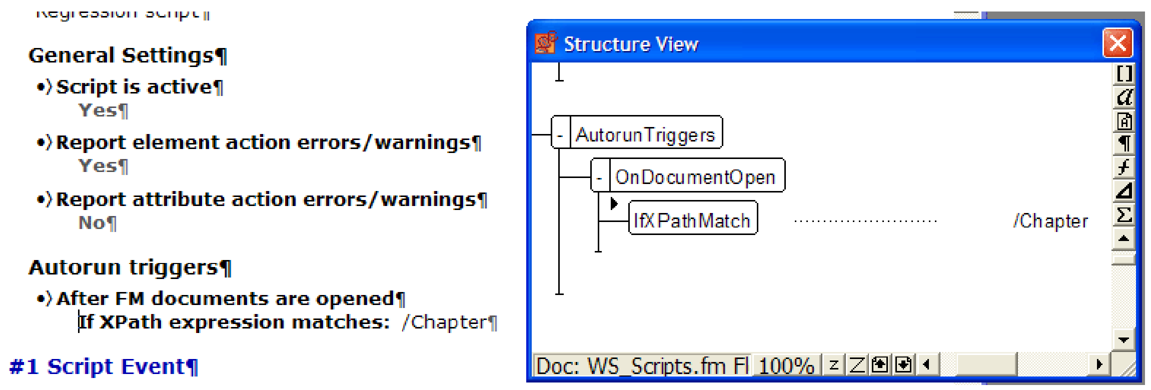
Autorun triggers

The script element may contain an optional `AutorunTriggers` element, which specifies when the script should automatically run. The default for each trigger is always “no,” and the absence of an `AutorunTriggers` element will disable all autorunning for the script.

Note: Script autorunning must be globally enabled in your FrameSLT preferences before any script will autorun. For more information, see *“Preferences”* on page 10.

Each subelement, such as `OnDocumentOpen` and/or `OnEDDImport` indicates a specific event that should trigger the script to run. There must be a `Yes` subelement to activate autorunning for the respective event. Alternatively, you may insert an `IfXPathMatch` element to control autorunning based on an XPath match. The text of `IfXPathMatch` should represent a valid XPath expression, and if the expression makes a single match on the respective document, the script will run at the respective trigger.

As an example, assume you have a script that you want to run when documents are opened, but only on documents that have a highest-level element of `Chapter`. Your script settings might appear as follows:



General Settings¶

- **Script is active¶**
  Yes¶
- **Report element action errors/warnings¶**
  Yes¶
- **Report attribute action errors/warnings¶**
  No¶

Autorun triggers¶

- **After FM documents are opened¶**
  If XPath expression matches: /Chapter¶

#1 Script Event¶

The following table describes the individual events for autorunning in more detail.

Element name	Event description
`OnDisplayRefresh`	After the screen is refreshed using the Ctrl+L shortcut.
`OnDocumentOpen`	After a binary FM document is opened. The results of the script will not be saved.
`OnEDDImport`	Following the import of element definitions through the menu path **File > Import > Element Definitions**.
`OnMarkupOpen`	After an XML, SGML, or MIF file is opened. The results of the script will not be saved.

Note the following important items about autorunning Node Wizard scripts:

- You should be very cautious when setting up scripts to autorun. It is easy to forget about them and you may find yourself wondering why strange things keep happening to your files, when it is actually a Node Wizard script running. Wherever possible, take advantage of the `IfXPathMatch` filter to restrict script autorunning to the desired files.
- Autorun scripts do not produce any warnings or error reporting, other than the element/action error report if the script specifies as such. If a script encounters an error that prevents it from proceeding, it will simply abort.
- Autorun settings apply to documents only. Scripts will not autorun on a book.
- If an event occurs that triggers both an autorun script and conditional text assignment with the conditional text management features, the conditional text assignment will occur first. For

more information on conditional text management features, see *"FrameSLT condition management"* on page 69.

- As with all Node Wizard script activities, no changes are saved afterwards. You can undo the effects of any script by closing the document without saving changes.

Event-level details

Script event settings define the actual work performed by the script when it is run. A script contains one or more events, each of which is wrapped in its own `NWScriptEvent` element. During a script run, FrameSLT steps through `NWScriptEvent` elements in order, performing actions as instructed by the event settings. If an event contains subevents, these events are also run as applicable before proceeding to the next event. The following sections describe the settings associated with a single event.

Disabling events and actions

In a script, all event and action container elements have a `Disable` attribute, which if set to "Yes", effectively "comments out" any subelements. All subordinate content, instructions, etc. should be completely ignored when running the script.

Event name and description

The event name (`EventName`) and description (`EventDescription`) are optional. If no name is provided, the event shows as "{no name}" in the scripts dialog. If no description is provided, the description box will be grayed out when the event is selected. Although you can specify this information for subevents as well, the scripts dialog box shows this information for top-level events only.

XPath expression

All events must have a valid XPath expression, represented as the text of an `XPathExpression` element. Expressions can start with a forward slash (/) to begin the query at the structural root or they can be contextual, depending upon the current context. In the case of top-level events, the current context is the current element/text selection in the active file. With this context-related support, you can run a script on a selected branch of a tree only. In all cases, you should always remain familiar of how your scripts are constructed and whether the current selection will have any effect on any given script.

Important note about context

If an XPath expression in a top-level event does not force the query to the highest-level element with a forward slash (/), the expression will depend upon the context of the current element selection. Therefore, when using scripts with these types of expressions, you should always be careful to set the initial context properly.

Limiting the number of matches

The `XPathExpression` element provides two attributes which you may use to control the extent of matching:

- `MaximumMatches` - Specifies the maximum number of nodes that you want the XPath to match, in document order. Zero or no value indicates to match all possible nodes.
- `SkipFirstMatches` - Specifies the number of matched nodes you want to skip before applying any actions and/or subevents. For example, if you specify 3 and the XPath matches 10 nodes, the actions and/or subevents will only be applied to nodes 4-10. Nodes 1-3 will be completely ignored. Zero or no value indicates normal behavior on all matched nodes.

If the two settings are used together, the count of maximum matches begins after the skipped matches. For example, if you only want to apply actions and/or subevents to the fifth matched node in document order, you would specify the following:

- `MaximumMatches` = 1
- `SkipFirstMatches` = 4

Sorting matches

As an option, you can place a `SortOptions` element directly following the `XPathExpression` element to sort the matches based on the content of the matched node. Sorting happens immediately following the XPath query and the new sort order remains permanent for the life of the event.

You can choose to sort:

- In ascending or descending order.
- Using numeric, alphabetic, or case-insensitive alphabetic order. Numeric sorting should be used if the content for all matched nodes represents real numbers, because alphabetic sorting of numbers can produce a numerically-incorrect order. For example, the following list would be considered correct for alphabetic order, but not numeric:
  - 1
  - 10
  - 2
  - 3

  If unspecified, the sort defaults to alphabetic.
- Based on another XPath query from the context of each match. That is, you can specify an additional XPath expression that is run against the context of each match to find another node from which to retrieve sortable content, rather than using the original matched node. If unspecified, the default is “`.`”, or in other words, sort based on the context of the original match.

Using parameters in XPath expressions

For XPath expressions in Node Wizard Scripts only, the use of parameters is supported. Before a parameter can be used, it must be properly set using the appropriate event action or an external call (see *““Other” actions”* on page 64 and `SetScriptParm` on page 140).

A parameter is effectively a variable. It must have some name and must be preceded by a dollar sign ($) when referenced, similar to W3C standards for parameters in XPath/XSLT.

In an XPath expression, a parameter can replace either a node name or a literal string. If the parameter follows an axis or is the argument of a `not()` function, it is assumed to be a node name. Otherwise, it is assumed to be a literal string. As a general guideline, if you want a parameter to be considered as a node name, always use an axis, even if you normally would not specify it explicitly. For example, if you intend to invoke the `child::` axis with a parameter, spell out the axis, for example, `child::$parm`.

The following tables shows some examples, using a theoretical parameter "`parm`" that has been set to "`Body`":

Sample expression	`//$parm`
Functional equivalent	`//Body`
Match behavior	Matches all `Body` elements in the tree.

Sample expression	`//Section/$parm`
Functional equivalent	`//Section/Body`
Match behavior	Matches all `Body` elements that are the children of `Section` elements.

Sample expression	`//Section[$parm]`
Functional equivalent	`//Section["Body"]`
Match behavior	Matches all `Section` elements in the file, because the parameter is regarded as simple literal string, which when presented alone, always returns a "true" condition.

Sample expression	`//Section[child::$parm]`
Functional equivalent	`//Section[Body]`
Match behavior	Matches all `Section` elements in the file that have a `Body` child. Note the use of the explicit axis to force consideration as a node name.

Sample expression	`//*=$parm`
	-or-
	`//*[.=$parm]`

Functional equivalent	`//*="Body"`
Match behavior	Matches all elements in the tree whose text content equals "Body".

Sample expression	`//$parm[.=$parm]`
Functional equivalent	`//Body[.="Body"]`
Match behavior	Matches all `Body` elements in the tree whose text content equals "Body".

Sample expression	`//*[contains(.,$parm)]`
Functional equivalent	`//*[contains(.,"Body")]`
Match behavior	Matches all elements whose text contains the string "Body".

Sample expression	`//*[contains($parm, "Body")]`

Functional equivalent	`//*[contains("Body", "Body")]`
Match behavior	Matches all elements in the tree, because the predicate tests the parameter value as a literal string.

Sample expression	`/*[$parm < "Busy"]`
Functional equivalent	`/*["Body" < "Busy"]`
Match behavior	Matches the HLE, because the string “Body” is lexicographically lesser than the string “Busy”. Note that this setup is useful for creating an “if” type of programming logic within a script; that is, an expression such as this can effectively be used to test the value of a parameter and control the implementation of actions afterwards.

Sample expression	`//*[contains($parm,"XRef")]`
Functional equivalent	`//*[contains("Body","XRef")]`
Match behavior	Matches nothing, because the predicate tests the parameter value as a literal string.

Sample expression	`//*[contains(child::$parm,"Sometext")]`
Functional equivalent	`//*[contains(Body,"Sometext")]`
Match behavior	Matches all elements that contain a `Body` child whose text includes the string “Body”. Note the use of the `child::` axis to force the consideration as a node name.

Sample expression	`fmfile::$parm`
Functional equivalent	`fmfile::Body`
Match behavior	Matches the HLE of a file named “Body”. Naturally, this is not a logical name for a file. The point is to show that parameters work with this axis as well.

Flows to process

An event may include a `FlowsToProcess` element, which indicates which flow(s) the event should process, either the main flow only or all structured flows in the document. In the absence of this element, the default behavior is to process the main flow only, unless an expression specifically designates a different flow (see *“Special flow-related axes”* on page 17). If the XPath expression is a contextual expression in a subevent, this setting is not relevant because all queries will automatically begin from the context passed down from the parent event.

Element actions

For each event, you can have zero or more `ElementAction` elements with the appropriate subelements to define element-based actions to occur for each match of the XPath expression. The action subelements follow the same naming convention as the Node Wizard action drop-down list and perform the same activities. When multiple actions are specified, they are attempted in the order shown in the script. For detailed descriptions of element action behaviors, see *“Element actions”* on page 39 and *“Element/attribute actions supported by scripts only”* on page 47.

In some cases, you need only insert elements to define actions for an event, because the element markup provides FrameSLT enough information. In other cases, you must enter additional information. The following table summarizes requirements for element actions in the scripts file:

Action element	Settings file requirement

`Copy_elem_contents_to_CB`
`Copy_elem_to_CB`
`Delete_element`
`Delete_contents`
`Demote`
`Move_down`
`Move_up`
`Promote`
`Unwrap`
	Only the action element is required. The action will occur on the matched element node(s), and no further information is necessary.

`Insert_elem_before`
`Insert_elem_after`
`Insert_first_child`
`Insert_last_child`
`Retag_as`
`Wrap_element_in`
`Wrap_contents_in`
	For actions that require a new element or a new element tag, use subelements of the action element to define what the tag should be. You can specify a tag directly or use other methods such as a parameter value or an XPath query for the tag name. Remember that the specified element must appear in the respective document’s EDD for these actions to work, and that element tags are case-sensitive.

Action element	Settings file requirement

`Paste_CB_after`
`Paste_CB_at_beginning`
`Paste_CB_at_end`
`Paste_CB_before`
`Paste_CB_over_contents`
`Paste_CB_over_elem`
	For actions that paste from the clipboard, the clipboard is handled as follows:
	• If the action element in the script contains absolutely no content, the action will paste the existing clipboard content without altering it. For this type of action to be useful, you must either populate the clipboard before running the script or combine “copy” actions with subevents to populate it during the script. For more information on subevents, see *“About subevents”* on page 46.
	• If the action element in the script contains any content, including a single whitespace, during processing FrameSLT will select the entire content of the action element and copy it to the clipboard for use with the specified element action. You may therefore use any content that can reside in a FrameMaker document, including text, elements, markers, anchored frames, etc.
	Note that the script settings file is a structured FM document, whose structure is defined by its internal EDD. Therefore, if you place content in any element that violates that EDD, it will appear as invalid. If you are using clipboard-based element actions to paste in structured content, it is very likely that you will have data in your scripts settings file that makes it invalid, simply because your target documents are unlikely to use the same EDD. For this reason, invalid content is acceptable within a script settings file for clipboard-based element actions.
	To place invalid structured content into the settings file, you will need to paste it, because you won’t be able to insert it with the element catalog.
	Note: Invalid content in the settings file will appear as red and will lack any expected formatting because the settings file has no instructions for formatting it. Once it is pasted into another document with a valid EDD, however, it should immediately assume the formatting you expect.

`Copy_elem_text_to_parameter`	Requires the specification of the parameter to capture the text.
`Set_elem_text`	Requires a subelement to specify the source of the text.
`Assign_conditions`	For conditional text assignment, you must enumerate each desired condition as the text of an individual `Condition` subelement. Keep in mind that condition names are case-sensitive, and should represent valid conditions defined in the template(s) on which the script will be run.

Attribute actions

For each event, you can have zero or more `AttributeAction` elements with the appropriate subelements to define attribute actions to occur for each match of the XPath expression. The action subelements follow the same naming convention as the Node Wizard action drop-down list and perform the same activities. For detailed descriptions of attribute action behaviors, see *"Attribute actions"* on page 41.

With all actions, you must specify some additional information with the action element, normally to indicate which attribute(s) and perhaps value(s) the action should be performed on. The following table summarizes requirements for attribute actions in the scripts file.

Note: If you use whitespace to delimit multiple values, a special general setting is required for the script to recognize that convention. For more information, see *"Script-level general settings"* on page 51.

Action element	Settings file requirement
`Copy_all_values_to_CB` `Copy_first_value_to_CB`	Only the action element is required. The action will occur on the matched attribute(s), and no further information is necessary.

`Add_specified_values`
`Remove_specified_values`
`Replace_values_with_spec`
	These actions require that you specify one or more attributes for the action and one or more values, represented as the text content of `Attribute` and `Value` subelements respectively. Keep in mind that attribute names and values are case-sensitive.
	`xpath-match` subelements are permitted under `Attribute` elements. See the note below.

`Delete_all_values` `Remove_invalid_attribute`	These actions require you to specify one or more attributes, but no values are necessary because values and attributes are deleted, not set. `xpath-match` subelements are permitted under `Attribute` elements. See the note below.

`Move_value_to_elem_text`
`Copy_value_to_elem_text`
`Move_elem_text_to_value`
`Copy_elem_text_to_value`
	These actions require a single `Attribute` subelement to specify which attribute the action should act upon. These actions are designed to operate with a single attribute only, with a single value.
	`xpath-match` subelements are permitted under `Attribute` elements, because these actions can act upon matched or specified attributes. See the note below.
	Note: These actions represent legacy functionality which is now possible using subevents. They are retained for backwards-compatibility, but may be deprecated in the future.

Action element	Settings file requirement

`Move_values_to_spec_attr`
`Copy_values_to_spec_attr`
`Move_values_from_spec_attr`
`Copy_values_from_spec_attr`
`Swap_values_with_spec_attr`
	These actions require a single `Attribute` subelement to specify which attribute the action should act upon. These actions are designed to operate with a single attribute only, with a single value.
	These actions always perform a transaction between a specified attribute and an XPath-matched attribute. Therefore, your XPath expression should be constructed to match attributes, and the specified attribute should appear under the action element. The `xpath-match` subelement is not applicable in this case, because the action itself already indicates that the matched attribute will be used for one side of the transaction. If you were to use `xpath-match` as the specified attribute, the action would have no effect because the transaction would attempt to occur between the matched attribute and itself.
	Note: These actions represent legacy functionality which is now possible using subevents. They are retained for backwards-compatibility, but may be deprecated in the future.

`Search_and_replace_string`	This action requires you to specify a search string (`SearchString`) and optionally a replacement string `(ReplaceString)`. If you do not specify a replacement string, all instances of the search string will be deleted. This action always occurs on XPath-matched attributes. Therefore, no attribute specification is required. When this action is performed in the Node Wizard, the dialog allows you to specify multiple search and replace strings. Although allowed, performing actions in this manner is discouraged, and the scripts setting file only allows a single `SearchString` and `ReplaceString` element. If you have multiple search and replace activities to conduct, use multiple events.

Action element	Settings file requirement
`Set_attr_value`	Requires a subelement to specify the source of the text.
`Paste_CB_to_matched_attr`	For actions that paste from the clipboard, the clipboard is handled as follows: • If the action element in the script contains absolutely no content, the action will paste the existing clipboard content without altering it. For this type of action to be useful, you must either populate the clipboard before running the script or combine “copy” actions with subevents to populate it during the script. For more information on subevents, see *“About subevents”* on page 46. • If the action element in the script contains any content, including a single space, during processing FrameSLT will select the entire content of the action element and copy it to the clipboard for use with the specified attribute action. For attributes, you should supply text content only. Elements, objects, and other such items are not applicable to attribute values.

Note: Under `Attribute` elements, you may type an attribute name, or you may insert an `xpath-match` element to indicate the attribute(s) matched by the XPath expression. For more information about specified attributes versus matched attributes, and details about attribute actions on matched attributes, see *“Attribute actions”* on page 41.

String operations

For each event, you can have zero or more `StringOperation` elements to define string operations to occur for each match of the XPath expression. A string operation consists of:

- One or more arguments (`Arg1`, `Arg2`, etc.) that define the parameters for the operation. The element catalog, in conjunction with automatic text in the scripts document, will guide you through the argument setup of an operation. Note that an argument can either be a static string or a parameter. If the specified value starts with a dollar sign ($), it is assumed to be a parameter. If you need to operate on a string that begins with a dollar sign, you must assign it to a parameter first, then call that parameter for the operation.
- A parameter to capture the result (`Result` element).

The following table briefly describes the supported operations. Note that their behavior generally follows the conventional logic used by most popular programming and scripting languages.

Note: All operations that use character indexes and/or string lengths are based strictly on the number of characters, which includes respect for multi-byte (Unicode-range) characters. The association between single bytes and single characters will not be reliable unless all strings contain ASCII characters only.

Operation element	Description
`Str_op_add_strings`	Adds (concatenates) two, three, or four separate strings. The third and fourth strings are optional.
`Str_op_get_length`	Retrieves the number of characters in the string.

Operation element	Description
`Str_op_index_of`	Retrieves the index of the first instance of the specified substring, or -1 if the substring does not exist. The operation includes arguments for: • The “start” index; in other words, the index from which to begin searching for the string. To search from the beginning, specify zero. • Case-sensitivity during the substring search, applicable to ASCII characters only and disabled (case-insensitive) by default.
`Str_op_replace`	Replaces all instances of a specified string with a specified replacement string. This operation includes the optional argument to specify case-sensitivity for the search (ASCII only), which is turned off (case-insensitive) by default. The ability to “clone” the case of the original string is not supported.
`Str_op_reverse`	Reverses the string, character by character. Note that this operation can produce certain anomalies with Unicode characters if code points are used to render a single character, such as the use of a diaeresis.
`Str_op_substring`	Retrieves the substring between the specified indexes.
`Str_op_rtrunc_by_index`	Removes (truncates) the portion of the string from the beginning to specified index, where zero is the position “before” the first character. For example: **Original string:**`The ΦΣΠ fraternity` **Specified index:** `5` **Result:** `ΣΠ fraternity`
`Str_op_trunc_by_chars`	Removes (truncates) the specified number of characters from the end of the string. For example: **Original string:**`The ΦΣΠ fraternity` **Specified chars:** `5` **Result:** `The ΦΣΠ frate`
`Str_op_trunc_by_index`	Removes (truncates) all characters following the specified index, where zero is the position “before” the first character. For example: **Original string:**`The ΦΣΠ fraternity` **Specified index:** `5` **Result:** `The Φ`

Numeric operations

The `NumericOperations` element currently supports a single operation, defined by a `Num_op_calculate_int_exp` (calculate integer expression) subelement. Within this subelement, you should provide an expression where all terms are integers and a parameter in which to capture the result.

Currently, expression support is rudimentary and includes the four major operators (+, -, /, and *) only. Additionally:

- A term may be a static integer or a previously-defined parameter.
- Multiple terms and operators may be provided.
- All terms and operators must be separated by whitespace.
- No operator “precedence,” parenthetical expressions, etc. are supported.

As an example, consider the following expression which will evaluate to 5, if parameter “`$integer` was previously defined as 10:

```
14 - $integer + 1
```

Because operation precedence is not supported, all operations follow a strict calculation path from left to right. For example, the following expression will evaluate to 9, rather than 7 as would be expected following normal precedence rules:

```
1 + 2 * 3
```

While this feature may have some use in its current state, it is mostly introductory and has much room for improvement, if there proves to be a need. If you have a need for improvement and/or expansion, we would like to hear from you.

“Other” actions

A script event can have zero or more `OtherActions` subelements, each of which can define zero or more advanced actions that are not necessarily related to element or attribute markup. The following table provides a brief description of currently-supported actions.

Action element	**Settings file requirement**
`Add_object` `Delete_object`	Allows the addition and deletion of table rows and columns. When feasible and applicable, it is recommended to use these actions rather than element-based actions that add and delete elements. To add or delete a row, the event XPath must match a row element or any subelement within. To add or delete a column, the XPath must match a cell element or any subelement within. ***Note:*** When deleting columns, it is recommended to delete one column per event and to have no other actions following the deletion. The deletion of a column can alter FrameMaker’s internal representation of the structure tree such that a new XPath query may be required to re-establish the proper context within the tree.
`Close_file`	Closes the file currently being processed. **WARNING!** The file will be closed as-is without saving changes! This action positively cannot be undone!

Action element	Settings file requirement
`Disable_screen_updates` `Enable_screen_updates`	Disables and enables screen updates during script processing. With screen updates disabled, scripts can operate much faster. In all cases, screen updates are restored following the termination of a script, whether expected or unexpected.
`Do_messaging`	Provides a set of interactive prompt options, including prompts that accept user input. Options are as follows: • `Do_Yes_No_Prompt` - A prompt with **Yes** and **No** buttons, with the initial focus on **Yes**. • `Do_No_Yes_Prompt` - A prompt with **Yes** and **No** buttons, with the initial focus on **No**. • `Do_OK_Cancel_Prompt` - A prompt with **OK** and **Cancel** buttons, with the initial focus on **OK**. • `Do_Cancel_OK_Prompt` - A prompt with **OK** and **Cancel** buttons, with the initial focus on **Cancel**. • `Do_OK_Prompt` - A prompt with an **OK** button only. The subelements are used to specify the text of the prompt. Additionally, for prompts that include a **Cancel** or **No** button, the behavior following a button click is as follows: • If a `Result` subelement is included, the value of the clicked button is assigned to the specified parameter, where **OK** or **Yes** causes the assignment of 1 (one) and **Cancel** or **No** causes the assignment of 0 (zero). Afterwards, the script continues processing normally. Note that you can use an XPath expression to conditionally evaluate which button was clicked and direct further processing as appropriate. For an example, see the samples that install with FrameSLT. -or- • If no `Result` subelement is included, **OK** or **Yes** causes the script to continue normally (that is, has no effect) and **Cancel** or **No** causes the script to abort immediately at that point. ***Tip:*** Because you can draw from the clipboard or a parameter for the message text, this action may be useful for script debugging.

Action element	Settings file requirement
`Do_messaging` (continued)	• `Do_path_browser_prompt` - A prompt with a file system browser tree that allows you to select a folder. • `Do_file_browser_prompt` - A prompt with a file system browser tree that allows you to select a file. For these prompts, if an item is selected and the user clicks **OK**, the value of the item is stored in the specified parameter (`Result`). In any other case, the string specified in your local preferences file is assigned to the parameter. Using factory settings, this string is `USER_CANCELED`. ***Note:*** To set the starting point for the browser, you can set the special “`NWS_StuffVal`” parameter with the file/path. When the browser prompt is launched, if this parameter contains a valid file/path, that location will become the starting point for the browse.
`Do_messaging` (continued)	`Do_string_entry_prompt` - A prompt with a field to enter a string. For this prompt, if the user clicks **OK**, the entered value is stored in the specified parameter (`Result`). This value may be an empty string. If the user clicks **Cancel**, the string specified in your local preferences file is assigned to the parameter. Using factory settings, this string is `USER_CANCELED`. Additionally, note the following: • Like all FrameMaker dialog boxes, the text input field will not process backslashes (\) properly, as these are used to precede certain escape sequences. If you require backslashes in the final value, consider requesting the use of forward slashes (or some other unique character) instead, then using a `Str_op_replace` action to replace the backslashes afterwards. • To prepopulate the text box, you can set the special “`NWS_StuffVal`” parameter with desired string. If this parameter is set, any value it contains becomes the default value for the text box.
`Exit_script`	Safely and immediately terminates script processing.

Action element	Settings file requirement
`Get_misc_data`	Copies all contents of this element (in the scripts file) to the current clipboard. If the element is empty, the action fails and the current clipboard is maintained. If the action fails, a warning is provided if “other” action warnings are enabled in the script `GeneralSettings` area.
`Get_object_property` `Set_object_property`	Allows you to retrieve or set a property of an underlying object. These types of actions are critical for advanced operations such as cross-reference generation and table sizing/formatting. When you get a property, the value is: • Assigned to a parameter, if an `Apply_to_parameter` element is inserted after the property element and contains a valid parameter name -or- • In any other case, copied to the clipboard When you set a property, you can set it based on clipboard text, parameter text, XPath-queried text, or static text entered in the scripts document. If you want to use the text content of the currently-matched node, use the XPath-queried text option with the simple “to-self” expression (`.`). The following properties are supported. Note that the current list can be expanded if necessary. Please contact West Street if you have a need for a property that is not currently supported. **Referenced graphic properties** (XPath must match a graphic element): `ImportObFile` - The filename or path of the referenced file. If the anchored frame contains multiple objects, a single object is selected at random. **Marker properties** (XPath must match a marker element): • `MText` - The full marker text • `MTypeName` - The case-sensitive marker type, such as “Index”

Action element	Settings file requirement
	Page-related properties (XPath can match any element or text frame: • `PageNumInt` - The absolute page number, with the first page starting at 1. • `PageNumStr` - The “formatted” page number, as assigned in the document numbering properties. This value may or may not start with 1 and may or may not be an integer, according to the document numbering properties. For example, if numbering is set to use Roman numerals, this property may retrieve values such as i, ii, iii, iv, etc. Note that if the match spans multiple pages, only the first page is retrieved. **Whole table properties** (XPath must match a table element or any subelement): `TblTag` - The case-sensitive table format **Table column properties** (XPath must match a table cell element or any subelement): `TblColumnWidth` - The width of the column in points **Cross-reference properties** (XPath must match a cross-reference element): • `XRefName` - The case-sensitive cross-reference format • `XRefSrcFile` - The filename or path of the document that contains the cross-reference destination. If the cross-reference is internal to the current file, this property returns an empty string. To set this property as an internal cross-reference, you can use the value “Current”. • `XRefSrcText` - The ID of the cross-reference destination; that is, the “ID Reference” value of the cross-reference element..
`Get_parameter` `Set_parameter`	Gets or sets a parameter value. When getting a value, it copies the value of the specified parameter to the clipboard. If the parameter is not set, the action fails and the current clipboard is maintained. A warning is provided if “other” action warnings are enabled in the script `GeneralSettings` area. When setting the value, you can choose a static value, an XPath-queried value, the value of another parameter, or any value that can be derived from the current clipboard content. Any previous value of the parameter is overwritten.

Action element	**Settings file requirement**
`Refresh_EDD`	Refreshes all EDD format rules in the context document.
`Save_file`	Saves the file currently being processed. **WARNING! This cannot be undone! In general, you should avoid the use of this action unless you are positive that the script is doing exactly what you intended! If the script does something unintended, those changes will be permanent!**
`Select_match`	Selects the currently-matched element in the document window and Structure View, as if you manually clicked the element in the Structure View. This action is typically most useful for debugging in conjunction with `Do_messaging`.
`Set_active_file`	Changes the current context document to the specified file, which is then passed to the next sibling event. You can use an absolute path or a filename, with forward-slashes permitted as filepath delimiters. This action should only be used at top-level events and allows you to override the context of the original file on which the script was run. Normally, the XPath expressions for all top-level events will automatically run against the file that was active when the script was run. With this action, that context file is changed for the remainder of the script. Note that this action is provided as a convenience for unusual circumstances. Normally, you can use the `fmfile::` axis to change the file context with more precision and reliability.
`Update_xrefs`	Performs a full-document cross-reference update on the context document, similar to selecting **Edit > Update References** in the FrameMaker interface. No errors or unresolved cross-references are reported.

FrameSLT condition management

Note: **This feature set is largely overlapped by Node Wizard scripts and is under consideration for deprecation.**

FrameSLT includes a comprehensive XPath-based utility for conditional text management. Using this utility, FrameSLT can automatically apply conditional text based on structural metadata during actions such as:

- Inserting and wrapping elements
- Editing attributes
- Opening documents
- Updating books

Because the conditional text management uses XPath, you can automatically associate conditions with your content based on element names, hierarchy, attribute values, and more. All condition assignment occurs at the element level, but the contextual evaluations may use any aspect supported by FrameSLT XPath. In many respects, this feature compensates for the absence of structure-based condition association in current EDD formatting capabilities.

Note: The dedicated condition management features have some overlap with the Node Wizard dialog and associated scripting capabilities, which also provide conditional text assignment based on structural markup. The important distinction is that the dedicated features are intended as a real-time, responsive authoring tool, while Node Wizard features are generally considered batch or post-processing activities. The two may be used in conjunction, but you may find it easier to manage your processes if only one or the other is used for conditions management.

Condition management settings

All condition management settings are accessed by selecting **FrameSLT > Condition Management Settings**. The following tables describe these settings in detail.

General condition management settings

The general settings apply globally to auto-conditionalization features, as follows:

Setting	Description
Automatically apply conditions after...	Causes FrameSLT to assign conditions as specified by the **Auto application settings** below, after the selected actions occur. Note that element and attribute actions cause condition application on the selected element, while the other actions will apply conditions throughout the whole document or book.

Setting	Description
Enable the following warnings...	Enables message box warnings, as follows: • **When a non-existent condition is found** This is the warning presented when FrameSLT attempts to apply a specified condition that does not exist in the current document. Note that this warning is always disabled for document- and book-wide condition application. • **When a condition is auto-created** This is the message presented when FrameSLT attempts to apply a non-existent condition, and then automatically creates it to complete the action. In the application settings below, you can enable auto-creation on an expression-by-expression basis. ***Note:*** In the editor, these settings use “three-state” checkboxes, which have an intermediate “half-checked” state. When half-checked, the warning is enabled for one occurrence only, after which it will not appear again during the current FrameMaker session.
When applying conditions, show all conditions first	Sets all conditions to be shown before attempting to automatically apply conditions. This setting is only applicable to document- and book-wide condition application. This setting is recommended, because auto-conditionalization on content with hidden conditional text can produce unpredictable results.

Auto application settings

The auto-application settings allow you to specify the XPath expressions for matching the elements to be conditionalized. Each expression contains its own independent set of parameters that affect its behavior, including the conditions to apply if the expression is matched.

Setting	Description
Expressions	XPath expressions for matching elements to be conditionalized. Attempted matching always occurs in the order the expressions are listed. And, each expression has its own independent set of conditionalization parameters which appear to the right. When expressions are added or edited, they are parsed for validity first. Invalid expressions cannot be used and are therefore not permitted. You can use the error report to help debug your XPath expressions. ***Note:*** Due to the internal processing model, all XPath expressions are automatically enclosed within a `self::*[]` expression internally. This extra portion is not visible in the editor, but will appear in the error report. If you use the error report to debug expressions, keep in mind that the expression will show this portion added. For examples of valid expressions, see *“Examples of expressions and settings”* on page 74.
Settings are active	Enables the selected expression. If disabled, the expression and associated settings are completely ignored by FrameSLT processing, but the settings will remain stored for later.
Clear existing conditions	If the selected expression is matched, causes FrameSLT to remove any existing conditions on the respective element before applying the specified condition(s). If this setting is not checked, FrameSLT will instead attempt to add the specified condition(s) to any existing conditions. ***Note:*** This setting is recommended, because adding conditions may be unreliable if existing conditions are not uniformly applied across the entire element.
Create conditions if necessary	If the selected expression is matched, causes FrameSLT to create any specified conditions that do not currently exist in the document. Auto-created conditions attempt to assume the color “Red,” if the color exists in the document. Otherwise, the new condition will have no condition indicator.

Setting	Description
Conditionalize parent element also	If the select expression is matched, causes FrameSLT to apply the specified condition(s) to the parent element as well. This setting is useful for auto-conditionalizing elements during element insertion that use EDD auto-insertions. When expressions are evaluated during element insertion, only the last-inserted element is evaluated. For example, assume you have a `Section` element that automatically inserts a `Heading` element. Also, assume that you would like `Section` elements to be auto-conditionalized upon insertion. Because the `Heading` element is always the last element inserted, it is the only element that will be evaluated. Therefore, you can use this setting in conjuction with a `Heading`-based XPath expression to auto-conditionalize `Section` elements. ***Tip:*** See *"Examples of expressions and settings"* on page 74 for an example.
Conditions	The conditions to apply if the selected expression is matched. These condition names must be specified exactly as they appear in your template, including case.

Processing details

When an element is auto-conditionalized, the expressions are evaluated in the order that they appear in the settings editor. FrameSLT will stop at the first expression that matches, if any, and ignore the rest.

When you invoke auto-conditionalization at a document level, each element is evaluated independently in logical order from the highest-level element through the ends of all branches. Document-level auto-conditionalization operates on all structured flows, including master and reference page flows.

Document- and book-wide actions

In the main **FrameSLT** menu, you can find commands for auto-conditionalizing throughout an entire document or book. The menu also includes a command for clearing all conditions. This command removes conditional text assignment only and does not delete any content or actual condition tags. If any content is hidden when this command is run, it is unaffected. This command removes all conditions assigned, whether by FrameSLT or not.

Note: Both commands operate on all structured flows.

Element-level actions

In your condition management settings, you can set FrameSLT to automatically conditionalize during element actions such as insertion and wrapping, and attribute editing. These actions affect the respective element only, and do not evaluate any descendant elements. For example, if you wrap several elements in a `Section` element, only the `Section` element is auto-conditionalized, with all child elements remaining unprocessed.

Tip: If you would like to auto-conditionalize an element and all descendant elements, use the **Apply Conditions** command in the right-click menus.

Examples of expressions and settings

In all cases, auto-conditionalization occurs at the element level, such as the element you just inserted, or the element currently under evaluation during a document-wide action. It always occurs one element at a time, with any given element evaluated independently from the context of itself. When an expression matches, the specified conditions are applied to that element, according the settings associated with that expression. Therefore, your XPath expressions should be set up to match some element, or perhaps multiple elements.

Note: The remaining discussion assumes some familiarity with XPath, which is necessary for the construction of auto-conditionalization expressions. If you are not familiar with the XPath standard, consider reviewing *"Chapter 2 About FrameSLT XPath"* on page 13 first.

All evaluations begin from the context of the element under evaluation. Therefore, the most basic method for matching elements is by name, using the `self::` XPath axis. For example, the following expression will match all Body elements, regardless of context or other factors:

```
self::Body
```

This expression says literally, "If I am myself, and my name is `Body`, then match." With this expression, any `Body` element in the document will match, and the associated conditions applied. For example, if your general settings specify auto-conditionalization during element actions, all `Body` elements will receive the specified condition(s) when inserted.

You can also use conjunctions within XPath expressions to denote multiple possibilities. For example, the following expression:

```
self::Body or self::BulletItem
```

...will match all `Body` and `BulletItem` elements.

Within the scope of FrameSLT XPath support, you can also use predicate node tests for detailed contextual evaluations. For example, the following expression will also match a `Body` element, but only if it has a `Product` attribute set to "MyProduct":

```
self::Body[@Product="MyProduct"]
```

The following table illustrates several more sample expressions.

Expression	**Description**
`self::Body or self::Para or self::Note`	Matches all `Body`, `Para`, and `Note` elements.
`self::Body and self::Para`	Matches no elements, because an element cannot be both a `Code` and a `Para` element.
`self::Body[@Product]`	Matches any `Body` element with a `Product` attribute, regardless of the attribute contents.
`self::*[@Product]`	Matches any element with a `Product` attribute, regardless of the attribute contents.
`self::Body[@Product="ProdA" or @Product="ProdB"]`	Matches any `Body` element with a `Product` attribute set to either "ProdA" or "ProdB".
`self::Body[@Product!=""]`	Matches any `Body` element with a `Product` attribute with any specified value.

Expression	Description
`@Product="ProdA"`	Matches any element with a `Product` attribute set to "ProdA".
`parent::Section`	Matches any element that is a direct child of a `Section` element.
`ancestor-or-self::Section`	Matches any `Section` element, and any element with a `Section` ancestor.
`ancestor::Table[@Output="Print"]`	Matches any element that is a descendant of a `Table` element, whose `Output` attribute is set to "Print".
`ancestor::*[@Output="Print"]`	Matches any element that has any ancestor element whose `Output` attribute is set to "Print".
`Body`	Matches any element that has a child `Body` element.
`self::Section[@Product="ProdA" or Body]`	Matches any `Section` element that has a `Product` attribute set to "ProdA", or has a child `Body` element.
`position()=2` - or - `2`	Matches any element that is at the second position on its respective branch. In other words, it is the second child of its parent.
`self::BulletItem[position() > 1]`	Matches any `BulletItem` element that is not the first element on its respective branch.
`self::*`	Matches any element.

For more information on FrameSLT and XPath, see *"Chapter 2 About FrameSLT XPath"* on page 13.

Important note about conditions management features versus the Node Wizard

With FrameSLT, you can manage conditions with the conditions management features and/or Node Wizard features, including Node Wizard scripting. It is important to note the differences between the two, such that you can make a practical decision about which is more appropriate for your situation.

If you require a conditions management tool that is author-focused and responds locally as edits are made to a document, the dedicated conditions management features may be more appropriate. The XPath expressions that the dedicated features use are always "current-element" focused, in that they always assume a “to-self (self::) starting context. Whether during authoring activities or full document-wide sweeps, all condition assignment focuses on a single element at a time, looking at its immediate context with regards to the specified XPath evaluation expressions. Therefore, condition management occurs more on an element-by-element contextual basis, which is a generally friendlier environment for authors.

Node Wizard activities, on the other hand, rely on overreaching XPath expressions to navigate processing throughout a document, and typically are used as a post-process during key events, such as publishing or saving a document. For example, you may typically have a single XPath expression that begins at the root of a document and uses “to-descendant” axes to query through a document and match nodes for processing. From a conditional text assignment perspective, you can get the same results with this type of processing, but the surrounding workflow and overall feel may differ from using the dedicated conditions management tools.

Tips on condition management

- If you use auto-conditionalization in any capacity, you should normally have all conditions showing at the time of auto-conditionalization. Otherwise, the results may be unreliable or unpredictable. For example, if you insert an element and it becomes conditionalized with a condition that is currently hidden, the results may be unexpected.
- The right-click menus for text and the Structure View include an **Apply Conditions** command, which auto-conditionalizes the selected element and any descendant elements.
- The expression `self::*` will match any element. You may choose to place this expression last in the list as a “catch-all” or default, perhaps to ensure that an element will be unconditional if no other expressions match. To use it in this capacity, you should specify to clear existing conditions, and leave the conditions list empty.
- If you want to make expressions more EDD-specific, you can construct them to consider a unique highest-level element or attribute. For example, assume that you have a particular structure definition that uses a `Chapter` HLE, and you want an expression to match elements only within that structure. An example expression might be:

  ```
  self::Body and ancestor::Chapter
  ```

  This expression will match `Body` elements, but only if the structure tree has a `Chapter` element somewhere in the ancestry.

Chapter 4 Transformations

PLEASE NOTE: The transformation features of the plugin are scheduled for deprecation. They currently remain active and are believed to work; however, they are no longer tested. Requests for technical support and bug fixes may be denied. As of FrameSLT 3.0, Node Wizard scripts have generally replicated this functionality and more. If you need assistance with migration, please contact West Street.

The FrameSLT transformation engine allows granular-level content reuse for structured documents. Conceptually, in many ways it emulates XSLT (Extensible Stylesheet Language - Transformation), a language standard developed and maintained by the W3C. A working knowledge of XSLT will help you understand FrameSLT, but is not necessarily required.

This chapter contains the following sections:

About FrameSLT vs. XSLT

XSLT, a transformation language designed and managed by the W3 Consortium, (www.w3.org), is a versatile standard for transforming XML documents into other text-based formats, such as HTML. While similar in concept to XSLT, FrameSLT has many significant differences, including:

- **Text files vs. WYSIWYG operation** With FrameSLT, all transformations happen within the FrameMaker interface, using structured FrameMaker files. Unlike XSLT, you can watch FrameSLT build documents as it happens, and the WYSIWYG interface allows convenient and comprehensive error reporting as problems occur.
- **Transformation element names** Many FrameSLT transformation elements are conceptually similar to XSLT transformation elements, but not all. Those that are have the same name as their XSLT counterparts, preceded by `FSLT_`. Others, however, are unique to FrameSLT and should not be assumed to have a counterpart within the XSLT standard.
- **Transformation element attributes** Like XSLT, FrameSLT relies on the attributes of transformation elements for the necessary processing instructions and metadata. And, as applicable, those attribute names are the same as their XSLT counterparts, such as `select` and `test`. Again, however, many FrameSLT transformation attributes are unique to FrameSLT.
- **Processing order** FrameSLT processes stylesheets in a simple top-to-bottom fashion, handling transformation elements as they are encountered. You can specify any source file at any time, and FrameSLT queries that document with the applicable XPath. However, XSLT is

somewhat different, in that you normally have a single stylesheet and source XML file, which are processed together in an possibly non-linear fashion. Because of the FrameSLT processing order, it does not support the same “templating” concept of XSLT, although it does have its own means of creating templates with the `FSLT_template` element, which operates differently. Also, the usage of the common XSLT XPath statement “/”, as in `<xsl:template match="/">`, has no relevance in FrameSLT. These differences will become more intuitive after you have used FrameSLT transformations a few times.

- **Input** The input stylesheets and source files for FrameSLT are all structured FrameMaker documents, not XML files. However, if you can get an XML file into FrameMaker first, you can transform it. FrameMaker can open any valid XML file and retain its structural qualities, making the content accessible to FrameSLT.
- **Output** The normal output of an XSLT processor is a text file, while the output of FrameSLT is always structured FrameMaker documents. Therefore, FrameSLT is generally catered towards content reuse purposes only. In contrast, a common usage of XSLT is to transform XML text files into HTML, which has no relevance within the structured FrameMaker environment. FrameSLT can be a powerful means of managing the content you ultimately wish to appear in HTML, but it cannot create the HTML for you.

Despite the differences, you should find that there are still many similarities, and a knowledge of one should help you understand the other.

Required steps to perform transformations

To perform transformations with FrameSLT, you must complete two prerequisite steps:

1. Customize your applicable EDD(s) to allow `FSLT` transformation elements. Transformations require these special elements, and FrameMaker does not allow the insertion of any element that is not defined in the EDD. For more information, see *“Customizing an EDD to allow transformation elements”* on page 79.

   Note: FrameSLT does provide a means of performing transformations without altering the EDD of the stylesheet, using unstructured “transformation markers.” However, this approach is generally not recommended for normal production work. For more information, see *“Using “FSLT_template” markers”* on page 86.

2. Import the customized EDD into your documents and build stylesheets for transformation.

Generally, the first step is performed once, after which you can build any number of stylesheets. The EDD of a stylesheet can be any EDD, as long as it contains the applicable transformation elements. Therefore, you can add the transformation elements to your current EDDs and use your existing documents as stylesheets.

Note: Because the initial step involves EDD alteration, you must have EDD development experience to complete it successfully.

About stylesheets and transformations

In FrameSLT, you set up stylesheets for transformation, much as you would with XSLT. These stylesheets may be any structured FrameMaker document, with any content. In your stylesheets, you can place transformation elements wherever the EDD allows, which FrameSLT can transform into other content as specified by those element types and attributes. Your stylesheets can contain any mix and match of “normal” content, and FrameSLT transformation elements.

When FrameSLT transforms a stylesheet, it walks through the element tree in a hierarchical fashion, starting at the highest-level element and moving through the ends of all branches. When it encounters a transformation element, it processes it as appropriate. All other elements are completely ignored. Therefore, you have complete flexibility when designing stylesheets.

Note: Currently, transformations occur in the main flow of the stylesheet document only. Support for other flows will be added based on user demand. If you have a need to transform flows other than the main flow, please contact West Street.

Transformation elements are prefaced by "`FSLT_`" and are controlled by your EDD, like any structured FrameMaker element. Therefore, you can use any EDD to develop stylesheets, provided that you have altered it to allow the applicable `FSLT` transformation elements. For more information, see *"Customizing an EDD to allow transformation elements"* on page 79.

Tip: In this document, the term "stylesheet" or "stylesheet document" is used frequently to denote the document that will undergo transformation. Keep in mind, though, that a stylesheet is nothing more than a structured FrameMaker document that allows transformation elements, and any structured document can be a stylesheet if it has the appropriate EDD.

Customizing an EDD to allow transformation elements

You may use any structured document with any EDD as a stylesheet. However, transformation activities do require the special library of `FSLT` transformation elements. Therefore, for transformation activities to take place, you must place definitions for these elements in your EDD(s), as needed to satisfy your transformation requirements.

`FSLT` element definitions are constructed no differently than any structured FrameMaker element. They are all Container-type elements with special names and attribute sets that FrameSLT recognizes and can use for transformations. Therefore, `FSLT` elements are added to an EDD like any other element. That is, you should:

1. Copy the applicable definitions into your EDD, by copy/pasting from the definitions file provided with FrameSLT, `XForm_Elements_EDD.fm`.

   Note: It is highly recommended that you copy and paste from `XForm_Elements_EDD.fm`, because a single typographical error in the definition may cause unexpected results at best, and crashes at worst. Transformation elements absolutely must have the exact attribute definitions found in the sample EDD.

2. Put provisions for these elements in the general rules of your existing elements, as applicable.

FrameSLT processes an `FSLT` element wherever it finds it, so you have complete flexibility over where to allow them, or even which elements to allow at all. If you have no need for certain `FSLT` elements, there is no reason to include them in your stylesheet EDD. For example, if you only have a need for `FSLT_copy-of` elements, and you only want your stylesheets to allow them within `Section` elements, you can add only the `FSLT_copy-of` definition and restrict it to the `Section` general rule. On the other hand, for broad, unrestricted `FSLT` element usage, you can copy all of the definitions into your EDD and use Inclusion rules at your highest-level elements to allow them virtually everywhere.

If you choose to include only certain `FSLT` elements, be aware that some elements also require others. For example, the `FSLT_choose` element requires the `FSLT_when` element, and perhaps the `FSLT_otherwise` element, to function. Or, to create a table with an `FSLT_table` element, you must also have all the other accompanying `FSLT` table component elements.

Notes: The concept of Inclusions is not supported by the XML standard. If you must export your stylesheets to XML, do not use Inclusions to incorporate `FSLT` elements.

For more information on which `FSLT` elements may require others to function, see the specific details for that element in *"Chapter 5 Transformation Element Reference"* on page 89.

Launching transformations

You can launch a transformation on the active book or document by selecting **FrameSLT > Transform**. Before transformation, consider the following:

- **Book vs. document transformations** A book transformation is nothing more than a series of document transformations, for each document file in the book. Keep in mind that transformations only affect the special transformation elements as they are found, so a document without any of these elements will be unaffected. For best results, consider opening all your chapter files before launching a book transformation.
- **Source file vs. duplicate file transformations** For documents and books, you can choose whether to transform the original files, or create duplicates first. If you choose to duplicate a document, the new document will receive the same name as the original with the text "(TRANSFORMED)" in the file name. If you duplicate a book, you must select an alternate folder to receive the duplicate book. In this case, the book and all its chapter files are duplicated, and all cross-references, graphic references, and other references are adjusted as appropriate to reflect the new path. File names are not changed during book duplications.

  Note: If you perform a transformation on your source files, be aware that these files may be significantly altered. Therefore, you should use this option with caution. For source file transformations, FrameSLT never saves them afterwards, so you can still close them *without saving changes* to restore your original files.

Editing transformation elements

When you insert a transformation element, FrameSLT produces an editor that allows the convenient input of the required attributes. You can also right-click on transformation elements in the document or structure view and select **Set Up FSLT Element** to reproduce the editor. While FrameMaker allows you to set attributes directly in the structure view, it is highly recommended that you use the FrameSLT editors instead. In many cases, the editors filter your options, making it less likely to input unusable parameters. Also, the editors perform important error checking that can help avoid critical errors during transformations.

Note: You should never edit a `source_file` attribute with the native FrameMaker attribute editor, because the FrameSLT editor performs important decisions about the specified file that FrameMaker's editor cannot. If you edit a `source_file` attribute with the FrameMaker editor, FrameSLT will likely be unable to find the source file.

Source file details

For most transformation elements that include an XPath expression for querying, you can specify any structured book or document as the source file for the query. If you specify a book, FrameSLT walks through the entire book until all matches are made, as applicable.

Whenever you directly specify a source file or book, your accompanying XPath expression must begin with the forward slash "go-to-root" axis, because FrameSLT needs the context of the root element to begin the query. However, if you select to simply "Inherit" the source file from an ancestor transformation element, the source file context cascades down with the XPath node context, and your XPath can be constructed without the initial slash. If you specify "inherit," there must be an ancestor transformation element somewhere that explicitly specifies a document or book. For more information on cascading contexts, see *"About cascading contexts"* on page 83.

Querying the "current" document

For most transformation elements with XPath, you can specify the "Current" document as the source file, meaning that the stylesheet will query itself. In this case, the respective transformation element will behave as normal, perhaps drawing content from somewhere else in the stylesheet. *Be cautious when using this specification,* however, because if your XPath happens to match the transformation element currently being processed, you may cause an endless loop or any nature of unexpected behavior. For example, if an `FSLT_copy-of` element happens to copy itself into its own output, your results will be, at a minimum, unpredictable.

If you choose to query the “current” document, you can optionally set the initial XPath context to begin at the transformation element itself. For more information on this feature, see *“About starting contexts”* on page 82.

Relative vs. absolute paths

In your preferences (see *“Preferences”* on page 10), you can set FrameSLT to prefer relative or absolute paths for transformation elements. If you choose relative paths but a relative path cannot be resolved, FrameSLT will use the absolute path. Relative paths are recommended if feasible for your implementation.

In your preferences, you can also set FrameSLT to attempt adjustment of all relative paths if the document is saved to a new location through the FrameMaker menus. This is a recommended setting as well.

Opening, closing, and saving source files

If necessary, FrameSLT will open any source files that are specified in transformation elements, in order to perform the respective query. In your preferences, you have the option to allow FrameSLT to close them again afterwards. For more information on preferences, see *“Preferences”* on page 10.

FrameSLT never saves changes to your source files! Therefore, if a transformation causes a change to a source file, FrameSLT *will not* close it, regardless of your preferences. In other words, if FrameSLT makes a change to a file that it opened, it expects you to review the changes yourself before committing the save.

There are two situations that may cause a change to a source file during transformation:

- **The source file is a chapter of the book undergoing transformation** If FrameSLT opens a source file for a query, and the file happens to be a chapter of a book undergoing the transformation, it will eventually be transformed itself. Because FrameSLT never closes files after a transformation (due to potential unreviewed changes), a file in this situation will not be automatically closed.
- **A cross-reference was created that targets an element in the source file** When cross-references are formed in structured Frame, the ID attribute of the target element must be populated. When FrameSLT creates a cross-reference, if the attribute is currently unspecified, it will populate the attribute itself. Therefore, if the target document were closed without saving the attribute change, the cross-reference would be broken afterwards. Because FrameSLT never saves changes to your source files, it cannot close a source file in this situation.

Use of parameters in source file paths

If your preferences are set to allow it, you can use parameters in source file paths (`source_file` attributes) on transformation elements. The following notes apply:

- Usage rules are similar to those described for parameters in XPath expressions (see *“About parameters in XPath expressions”* on page 85), except that file paths are not parsed. The consequence for improper usage of parameters in file paths is simply opening the wrong file or a failure to find the file at all.
- Transformation element dialog boxes do not allow manual editing of source file paths. You must use the normal attribute editor to add parameters. Note that you should use forward

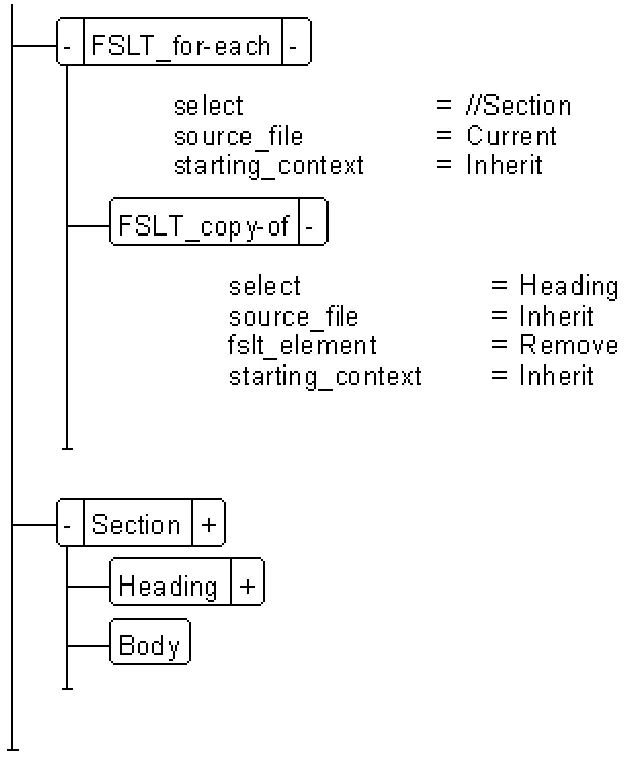
slashes (/) as file separators instead of backslashes, because backslashes are considered escape sequences and can be difficult to enter in an attribute editor.

About starting contexts

For most transformation elements with an XPath expression, if you choose to query the “current” document, you can also choose where to set the initial context for the XPath query:

- **Specified/Inherit** The XPath begins at the inherited context, or at the root if the XPath begins with a forward slash (‘/’). For more information on inherited contexts, see *“About cascading contexts”* on page 83.
- **FSLT element** The starting context is set at the transformation element itself, and any inherited context is ignored. If the XPath begins with a forward slash (‘/’), the root becomes the context regardless.

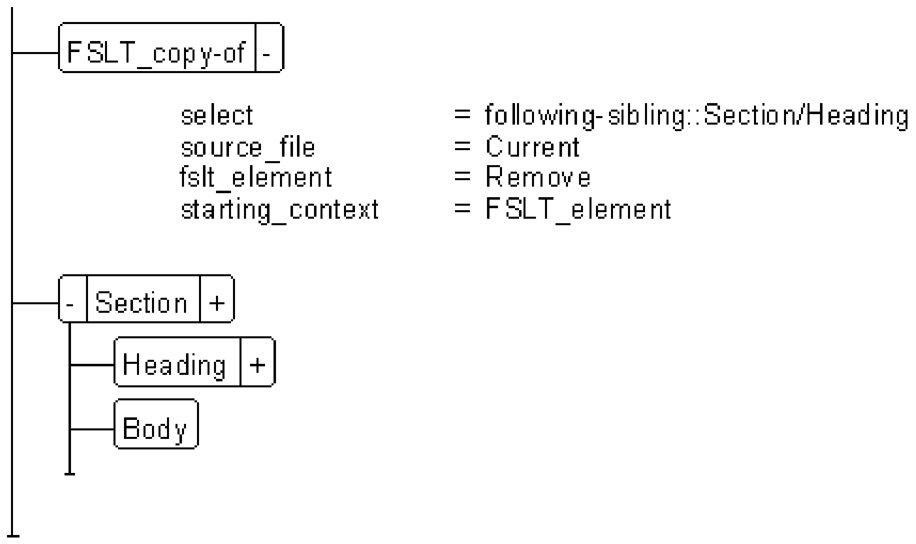
As an example, consider the following element tree, with two transformation elements:



```
- FSLT_for-each -
      select              = //Section
      source_file         = Current
      starting_context    = Inherit
  FSLT_copy-of -
      select              = Heading
      source_file         = Inherit
      fslt_element        = Remove
      starting_context    = Inherit
- Section +
  Heading +
  Body
```

Both transformation elements have the `starting_context` attribute set to “Inherit.” The `FSLT_for-each` element is set to find all `Section` elements in the current document. Then, the `FSLT_copy-of` element will pick up on the respective `Section` context and find `Heading` children. Therefore, the `Heading` element shown below will eventually be matched during some iteration of the `FSLT_for-each`/`FSLT_copy-of` combination, but not in any relation to the proximity of the transformation elements themselves.

Conversely, consider the following setup:


FSLT_copy-of
select = following-sibling::Section/Heading
source_file = Current
fslt_element = Remove
starting_context = FSLT_element
Section
Heading
Body

In this case, the `FSLT_copy-of` element starting context is specifically indicated as "FSLT_element." This means that the starting context of the XPath expression,

```
following-sibling::Section/Heading
```

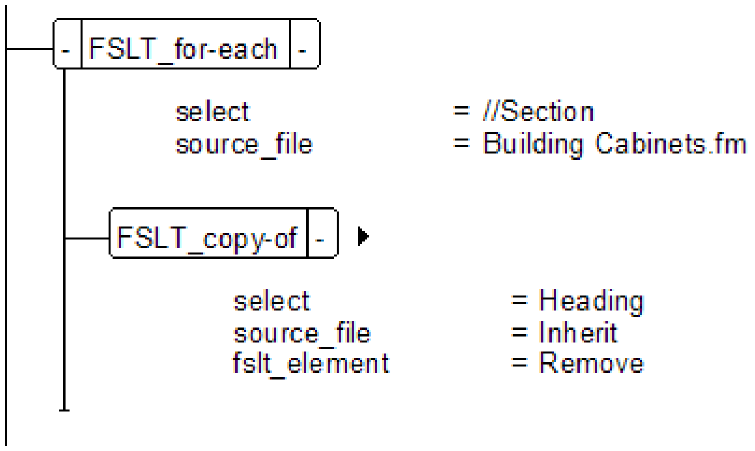
...will be the `FSLT_copy-of` element itself, in which case the first match will be the `Heading` element seen below. Or in other words, the XPath expression means literally, "Find any `Heading` elements that are children of following-sibling `Section` elements, beginning at the specified context." Because the specified context is the `FSLT_copy-of` element, the XPath walks straight down and matches the `Heading` element.

The starting context feature of FrameSLT opens up powerful possibilities for a stylesheet to manipulate its own structure, based on its own contents. In particular, you have significant flexibility with the `FSLT_create-xref` element to create detailed cross-reference structures for navigational aids such as inner-file tables-of-contents and "breadcrumbs."

Note: Because the ability for a stylesheet to query itself is unique to FrameSLT, versus XSLT, the concept of starting contexts is also unique. The attribute and its functionality have no counterpart in XSLT.

About cascading contexts

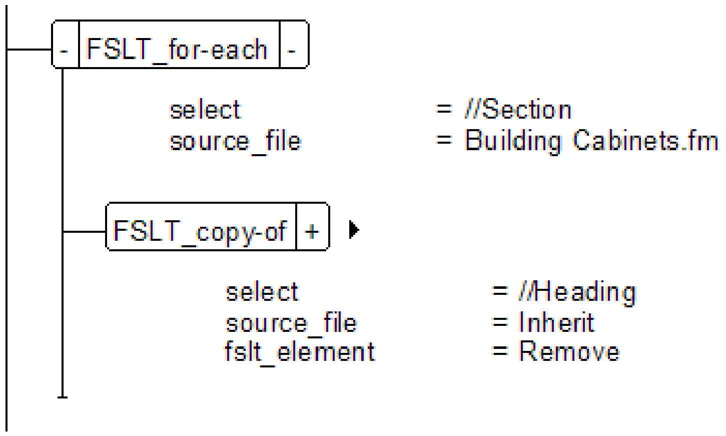
Like XSLT, contexts from FrameSLT transformation element matches are passed down to all descendant transformation elements. Therefore, if you have subordinate transformation elements that use XPath, that XPath can assume the context last set by the ancestor element. For example, consider the following two transformation elements:



```
- FSLT_for-each -
    select          = //Section
    source_file     = Building Cabinets.fm
  FSLT_copy-of -
      select        = Heading
      source_file   = Inherit
      fslt_element  = Remove
```

Assume that `Building Cabinets.fm` consists of `Section` elements, each with a single `Heading` child. In this case, the `FSLT_for-each` element matches the first `Section` element and passes that context down to the `FSLT_copy-of` element. The `FSLT_copy-of` XPath, then, starts there and matches all `Heading` children of that `Section`, which should be only one.

At any time, you can reset the node context by beginning your XPath with a forward slash ("go-to-root"). For example, consider the following variation:



```
- FSLT_for-each -
    select          = //Section
    source_file     = Building Cabinets.fm
  FSLT_copy-of +
      select        = //Heading
      source_file   = Inherit
      fslt_element  = Remove
```

In this case, the XPath of the `FSLT_copy-of` begins with a forward slash, meaning that it will ignore the context passed down by the `FSLT_for-each` and begin anew at the root. This particular XPath, in fact, will locate and copy over all `Heading` elements in `Building Cabinets.fm`.

In the previous examples, you can also see the cascading nature of the source file context. The child `FSLT_copy-of` element begins its query on the same source file as its parent transformation element. If the top-level source file were a book, the `FSLT_copy-of` element would assume the context of whichever chapter file contained the match made by the parent `FSLT_for-each` element.

At any time, you can break the source file context and specify a new source file. However, if you do, the associated XPath must begin with a forward slash (go-to-root). Without the go-to-root context, FrameSLT would have no context by which to begin the query in the new document or book.

About preserving transformation elements after a transformation

On an element-by-element basis, the following transformation elements can be set to remain in a post-transformed document:

- `FSLT_copy-of`
- `FSLT_create-xref`
- `FSLT_param`
- `FSLT_set-attribute`
- `FSLT_set-marker`
- `FSLT_template`
- `FSLT_value-of`

If set for preservation, the respective element remains in the transformed document with the original settings, and can undergo transformations repeatedly. In the case of `FSLT_copy-of`, `FSLT_value-of`, `FSLT_create-xref`, and `FSLT_template`, any content copied into the stylesheet becomes the child(ren) of the transformation element. To facilitate this action, all contents of these elements are automatically cleared at the beginning of transformations. In the case of the other two, the body content of the stylesheet is never affected, so they can simply remain as is.

The ability to preserve these elements provides powerful possibilities for repetitious content reuse and regular updates. For example:

- **Text inset replacement** If you use text insets, and your source text is always from structured FrameMaker documents, you could use `FSLT_copy-of`, `FSLT_value-of`, and `FSLT_template` elements as a superior replacement. With these elements set to be preserved, you could run repeated transformations on your source files to update the “inset” text at any time, after which the content is inserted directly into your document. The text would be completely editable like any other content, except that you could overwrite it with an update at will.

  Tip: The sample files included with FrameSLT contain many setups that demonstrate this functionality.

- **Automatic “breadcrumb” and TOC generation** If you use breadcrumbs that follow a certain structural pattern or logic, you can use `FSLT_create-xref` elements to completely automate the process and allow updates at any time. For example, assume that all your “Level 1” sections should have a cross-reference list to all subordinate “Level 2” sections. In a situation such as this, the XPath-based cross-reference generation provided by FrameSLT can significantly enhance the navigability of your documents, at a fraction of the time required to do it manually.

  Tip: To see an example of breadcrumbs automatically created by FrameSLT, see the cross-reference list at the beginning of this chapter.

If you want to use FrameSLT in this fashion; that is, running repeated transformations on the same source files, please note that *all* transformation elements in the files must allow preservation, and be set up as such. Otherwise, your stylesheet will be different after the first transformation, and a repeat transformation will not produce the same results. For example, if you have an `FSLT_for-each` element in the stylesheet, it will be removed after the first transformation regardless, and subsequent transformations will not consider it.

About parameters in XPath expressions

If enabled in your preferences, FrameSLT supports the use of parameters in XPath expressions for transformation elements. The following notes apply:

- Before it can be resolved in an expression, a parameter must be defined with an `FSLT_param` element. Unresolved parameters will cause a parsing error and a transformation to abort.
- A parameter is indicated with a dollar sign ($). Whenever a dollar sign is encountered, FrameSLT combines the following characters one-at-a-time until the resulting string matches a known parameter. If the end of the expression is reached before a match is made, the parameter is considered unresolved.
- Whenever a parameter is used in an expression, the expression cannot be parsed until the transformation process actually reaches that element. Therefore, parsing errors may occur in the middle of a transformation depending upon the value of the parameter at that time. For any transformation elements that do not use parameters, the expressions are parsed before the transformation process begins.
- Parameters are allowed anywhere in an expression, including within string literals if your preferences are set to allow it. This freedom of usage provides considerable flexibility but also adds a burden of responsibility for a stylesheet developer. A small error in an expression with parameters or an unexpected parameter value at the time of transformation can have significant consequences.
- When a parsing error occurs, the error report will show the expression with the parameter(s) replaced by the respective value(s). You must refer to the original stylesheet to see the original expression with the parameters.
- For external calls only, parameter values can be pre-defined before transformation (analogous to passing parameters to an XSLT stylesheet). For more information, see *“`SetParam`”* on page 139.

As an example, consider the following expression:

```
//$MyParameter
```

If there were a parameter named “MyParameter” defined as “Heading” at the time of transformation, FrameSLT would attempt to parse the following expression:

```
//Heading
```

Or, if there were a parameter named “Param” defined as “Heading” at the time of transformation, FrameSLT would attempt to parse the following expression:

```
//MyHeadingeter
```

For more information on preferences, see *“Preferences”* on page 10.

Using “FSLT_template” markers

Normally, transformation of a document requires that its EDD contain the necessary transformation elements to perform the desired tasks. However, FrameSLT does provide an alternative using unstructured “FSLT_template” markers. With these markers, you can call in transformation “templates” from another document and transform any stylesheet, regardless of the stylesheet’s own EDD.

To use “FSLT_template” markers, you should understand how the `FSLT_template` element works first. The concepts are similar, and explained in more detail in *“`FSLT_template`”* on page 112

Before attempting to use “FSLT_template” markers, please note the following:

- The best way to set up stylesheets and perform transformations is to adjust the EDD accordingly and use transformation elements instead. Using “FSLT_template” markers can allow powerful and comparable transformations, but the logistics of their use are much less flexible.
- Transformation elements are required for transformations, regardless of how they are introduced into the stylesheet. Therefore, to perform transformations on a stylesheet whose EDD does not define transformation elements, the “FSLT_template” marker must copy in

transformation content from a document whose EDD does. Therefore, you must ultimately have some document, somewhere, that allows you to create transformation setups using the `FSLT_template` element.

- Marker-based transformations are always a one-time transformation; that is, you cannot re-transform the same document more than once, unless you are creating duplicates each time.
- Your sample files include an example that performs a marker-based transformation. You may find that an examination of this file to be the best way for understanding how these markers work.

How “FSLT_template” markers work

When FrameSLT encounters an “FSLT_template” marker, it acts similarly to an `FSLT_template` process, in that it looks for another `FSLT_template` element with the same template ID, and copies its contents into the stylesheet. In the case of the marker, the template ID is specified as part of the marker’s text, instead of an attribute on a structural element. As with `FSLT_template` elements, you can direct the search towards an “FSLT_template” flow in the current document, or another document altogether.

Because a “FSLT_template” marker causes content to be copied into the stylesheet, but lacks any structural hierarchy to manage the incoming content, the process cannot be performed twice on the same stylesheet. That is, the content will be copied in at the location of the marker, but FrameSLT will have no means of identifying that content during any future transformations. Therefore, “FSLT_template” markers cannot facilitate any kind of “refreshable,” text inset-type architecture.

An “FSLT_template” marker can copy any nature of content into the stylesheet, including transformation elements. If the stylesheet’s EDD does not provide for transformation elements, any transformation elements will be invalid after insertion, but will be processed like any other transformation element. FrameSLT does not require a transformation element to be valid in order to process it.

For more information on how to set up an “FSLT_template” marker, see *“Adding markers to the stylesheet”* on page 87. For more information on how the `FSLT_template` element works, including information on “FSLT_template” flows, see *“`FSLT_template`”* on page 112.

Creating the marker type

To put any type of marker in a document, that type must be defined in the document’s template. In this case, you must create an “FSLT_template” marker, if it does not already exist. To create an “FSLT_template” marker type, refer to the FrameMaker help documentation. When creating the marker type, note the following:

- You must adhere to the exact spelling and case of the marker type, “FSLT_template.”
- You do not need to alter the EDD, even for the markers themselves. “FSLT_template” marker functionality is designed to work with unstructured markers.

Adding markers to the stylesheet

When adding the markers, you should insert them like any marker, selecting “FSLT_template” as the type. For the marker text, you must enter the following:

```
[TemplateSourceDoc]---[TemplateID]
```

where:

- **`TemplateSourceDoc`** is the name of the document where the corresponding `FSLT_template` element is located. If you specify “Current,” FrameSLT will search the “FSLT_template” flow of the current stylesheet, if it exists.

Note: If you specify an external document, you must specify a document file name only, and the file must be in the same folder as the stylesheet. With "FSLT_template" markers, FrameSLT cannot search any files outside of the current folder, nor can it search whole books.

- **`TemplateID`** is the ID of the target `FSLT_template` element, specified in the `template_ID` attribute.
- **`---`** is the required delimiter between the two arguments. It must be exactly three dashes with no spaces on either end.

The following are some examples of "FSLT_template" marker text:

`MyTemplatesDoc.fm---Template1`

(Searches `MyTemplatesDoc.fm` for an `FSLT_template` element with "Template1" specified for the `template_ID` attribute. It searches the "FSLT_template" flow first, if it exists, then the main flow.)

`Current---Template2`

(Searches the current document within the "FSLT_template" flow, if it exists. Note that a current document's EDD controls element availability in all flows, so it is not likely that you would use a marker in this case. If your EDD allows transformation elements, you should put them directly in the main flow, rather than using markers to call them from the "FSLT_template" flow.

Be conscious of the location when inserting "FSLT_template" markers. An unstructured marker can be placed nearly anywhere, but if you place it in a location that obstructs content from being copied in, such as within an <EMPTY> container element, the process may fail. FrameSLT attempts to insert the content at the exact location of the marker, which may present an architectural challenge because it can be difficult to ascertain the exact "structural" location of an unstructured marker.

Chapter 5 Transformation Element Reference

PLEASE NOTE: The transformation features of the plugin are scheduled for deprecation. They currently remain active and are believed to work; however, requests for technical support and bug fixes may be denied. As of FrameSLT 3.0, Node Wizard scripts have generally replicated this functionality and more. If you need assistance with migration, please contact West Street.

This chapter contains detailed information on each transformation element, including required parameters and processing specifics. For general information on transformations, see *“Chapter 4 Transformations”* on page 77.

Transformation elements supported by FrameSLT include:

- *“`FSLT_choose`”* on page 89
- *“`FSLT_copy-of`”* on page 91
- *“`FSLT_create-xref`”* on page 93
- *“`FSLT_for-each`”* on page 95
- *“`FSLT_if`”* on page 97
- *“`FSLT_otherwise`”* on page 99
- *“`FSLT_param`”* on page 99
- *“`FSLT_set-attribute`”* on page 100
- *“`FSLT_set-marker`”* on page 102
- *“`FSLT_sort`”* on page 104
- *“`FSLT_table`”* on page 106
- *“`FSLT_tablebody`”* on page 110
- *“`FSLT_tablecell`”* on page 110
- *“`FSLT_tableheading`”* on page 111
- *“`FSLT_tablefooting`”* on page 111
- *“`FSLT_tablerow`”* on page 111
- *“`FSLT_tabletitle`”* on page 112
- *“`FSLT_template`”* on page 112
- *“`FSLT_value-of`”* on page 116
- *“`FSLT_when`”* on page 119

FSLT_choose

The `FSLT_choose` element allows you to set up a structure that makes any number of XPath-based evaluations, stopping at the first one that holds true and performing the directed tasks. The operation is conceptually similar to that of `FSLT_if`, except that you can set up multiple conditions.

FSLT_choose processing

FSLT_choose requires one or more child FSLT_when elements, which is where the evaluations take place. During transformation, FrameSLT steps through the FSLT_when elements in order, testing the XPath for each one. If one matches (that is, the XPath finds something), the content of that FSLT_when element is added to the stylesheet. After a match, no further evaluations are made, and all other content of the FSLT_choose element is simply removed.

FSLT_choose can optionally include an FSLT_otherwise element at the end, as a default if all previous evaluations prove false. The content of an FSLT_otherwise is always added to the stylesheet if all FSLT_when evaluations fail. If an FSLT_when evaluation holds true, however, the FSLT_otherwise is discarded like the rest of the FSLT_choose content.

FSLT_choose, FSLT_when, and FSLT_otherwise elements never retrieve content from the source files themselves. However, you can use descendant transformation elements to retrieve content based on a new or inherited context. For more information on cascading contexts, see *"About cascading contexts"* on page 83.

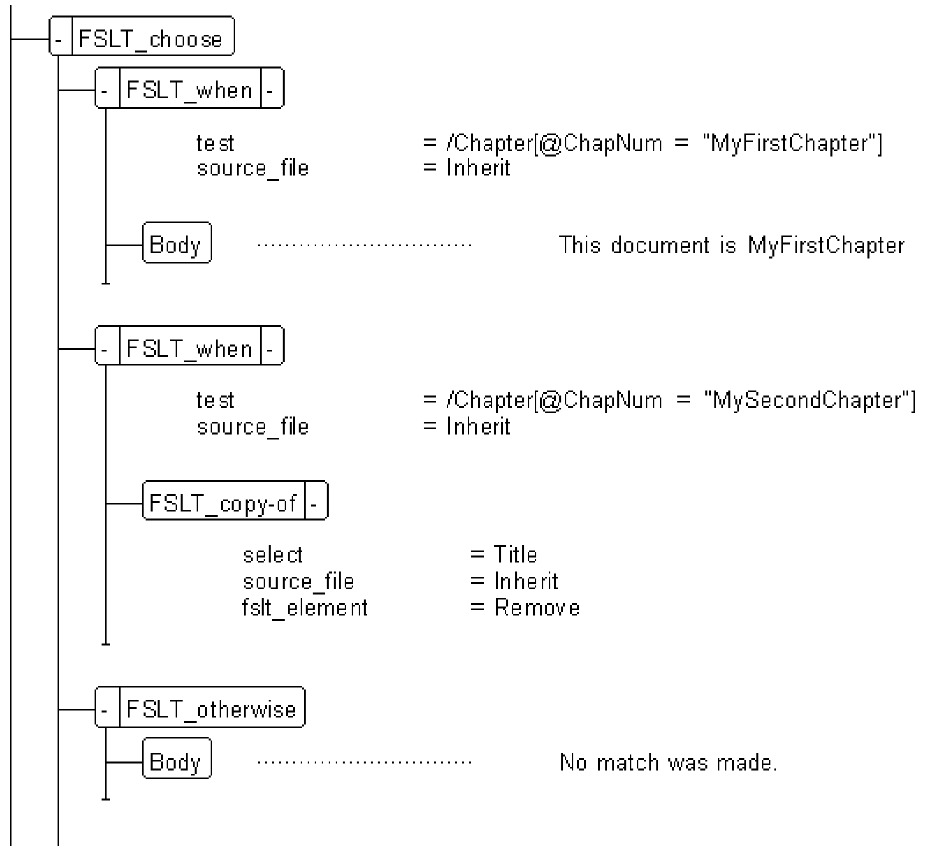
Note: In a normal XSLT environment, an xsl:when element itself should not pass any context to descendant transformation elements, unlike elements such as xsl:for-each which do pass down the context established by the XPath match from the select attribute. That is, if the XPath expression contained in the test attribute of xsl:when does make a match, it is for testing only and the context of the match does not get passed down. However, previous to version 2.0, FrameSLT erroneously did pass down the context from an FSLT_when XPath match. In an effort to fix this problem while maintaining backwards compatibility, the FrameSLT preferences now include an option to process in either fashion. For more information, see *"Preferences"* on page 10.

FSLT_choose attributes

FSLT_choose has no attributes that you need to set. All XPath-based and other attributes are specified at the child FSLT_when element(s).

FSLT_choose example

The following figure illustrates a sample FSLT_choose attribute structure:



```
- FSLT_choose
  - FSLT_when -
      test           = /Chapter[@ChapNum = "MyFirstChapter"]
      source_file    = Inherit
    Body  ..............................  This document is MyFirstChapter
  - FSLT_when -
      test           = /Chapter[@ChapNum = "MySecondChapter"]
      source_file    = Inherit
    FSLT_copy-of -
        select         = Title
        source_file    = Inherit
        fslt_element   = Remove
  - FSLT_otherwise
    Body  ..............................  No match was made.
```

During transformation, the following events occur:

1 For the context source file, if the `ChapNum` attribute of the highest-level `Chapter` element is set to “MyFirstChapter”, the `Body` element with the text “This document is MyFirstChapter” is added to the stylesheet.

2 Else, if the `ChapNum` attribute of the highest-level `Chapter` element is set to “MySecondChapter”, the `FSLT_copy-of` element is added to the stylesheet. The `FSLT_copy-of` element is subsequently processed, resulting in any child `Title` elements being copied to the stylesheet.

3 Else, if neither `FSLT_when` element makes a match, the `Body` element contained by the `FSLT_otherwise` element is added to the stylesheet.

Following transformation, all `FSLT_choose`, `FSLT_when`, and `FSLT_otherwise` elements are removed from the stylesheet, leaving only the contents of the applicable element for which a match was made, if any.

FSLT_copy-of

`FSLT_copy-of` is one of the primary elements for retrieving content from your source files. It performs an XPath query for elements and copies over any that match the XPath.

FSLT_copy-of processing

`FSLT_copy-of` operation is basic. It queries your source files based on the specified XPath, and for any elements that it matches, it copies them to the stylesheet. The copy includes all child elements and text. It continues copying over elements until all matches are exhausted, within the scope of the “max_matches” attribute.

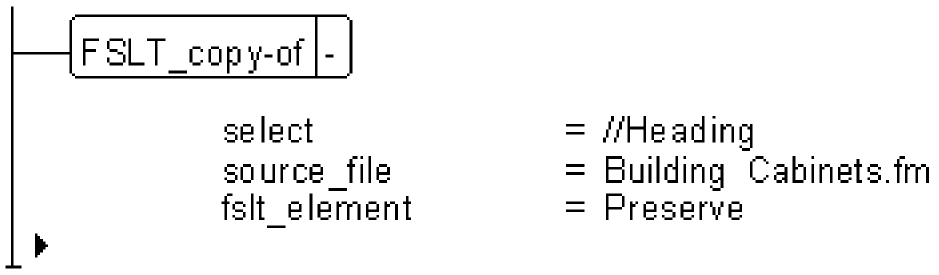
Because the content retrieval is element-based, your XPath should not search for attributes. If the final axis of an `FSLT_copy-of` XPath matches an attribute, the transformation will abort and return an error. Also, certain elements cannot be copied independently, such as table components. An attempt to copy one of these elements to your stylesheet will also cause the transformation to abort.

FSLT_copy-of attributes

Attribute	Description
`select`	XPath expression for the query. Any element matched will be copied to the stylesheet.
`source_file`	Source file or book for the XPath query. For more information on source files, see *"Source file details"* on page 80.
`fslt_element`	Whether or not to preserve the `FSLT_copy-of` element following a transformation. For more information, see *"About preserving transformation elements after a transformation"* on page 85.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the "current" stylesheet or document. For more information, see *"About starting contexts"* on page 82.
`max_matches`	The maximum number of matches permitted for this element, with zero (0) indicating unlimited (match all).

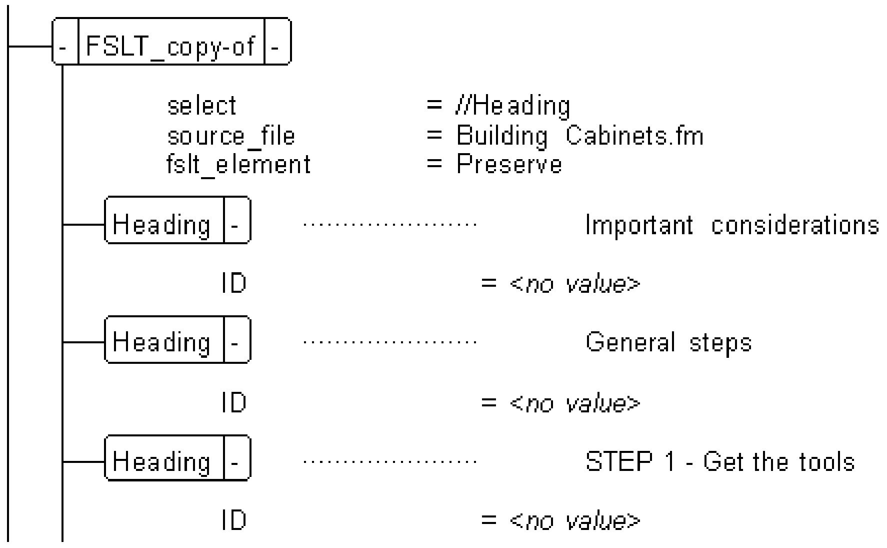
FSLT_copy-of example

The following figure shows an actual element from the sample file `Sample1_CopyOf`, which is included with FrameSLT:


FSLT_copy-of -

```
select          = //Heading
source_file     = Building Cabinets.fm
fslt_element    = Preserve
```

This `FSLT_copy-of` element is configured to look for all `Heading` elements in the `Building Cabinets.fm` file. For any that are found, they are copied to the stylesheet document. Following transformation, the element structure appears as follows:



The `FSLT_copy-of` element still appears in the transformed document, because it was set to be preserved. Had it been set to be removed, the results would have been the same, except that the new `Heading` elements would be on the main branch, and the `FSLT_copy-of` element would be gone.

To see this particular transformation occur, open `Sample1_CopyOf.fm` and run a transformation.

FSLT_create-xref

An `FSLT_create-xref` element allows you automatically create one or more cross-references, based on an XPath query.

FSLT_create-xref processing

`FSLT_create-xref` performs a simple XPath query of any supported source file or book, and creates a cross-reference to each element that is matched. It continues creating cross-references until all matches are exhausted.

`FSLT_create-xref` attributes include parameters about element tags and formats for the generated cross-references. You can also optionally choose an element to wrap each generated cross-reference.

Because `FSLT_create-xref` generates new cross-references, the transformation process requires a full-document cross-reference update following a transformation, in order to populate the new cross-reference text. All cross-references in your document, not just those generated, will be updated during this process.

Because cross-references always point to elements, not attributes, your XPath should search for elements, not attributes. If your XPath does locate attributes, the resulting cross-references will simply point to the elements where the attributes were found.

Special note on generated cross-references

All cross-references generated by `FSLT_create-xref` are element-based, versus the marker-based type used in unstructured FrameMaker. Therefore, the source of any generated cross-reference must have an `ID` attribute designed as a “Unique ID” type. In other words, the XPath of an `FSLT_create-xref` must match elements with `ID` attributes, otherwise FrameSLT cannot establish the cross-references.

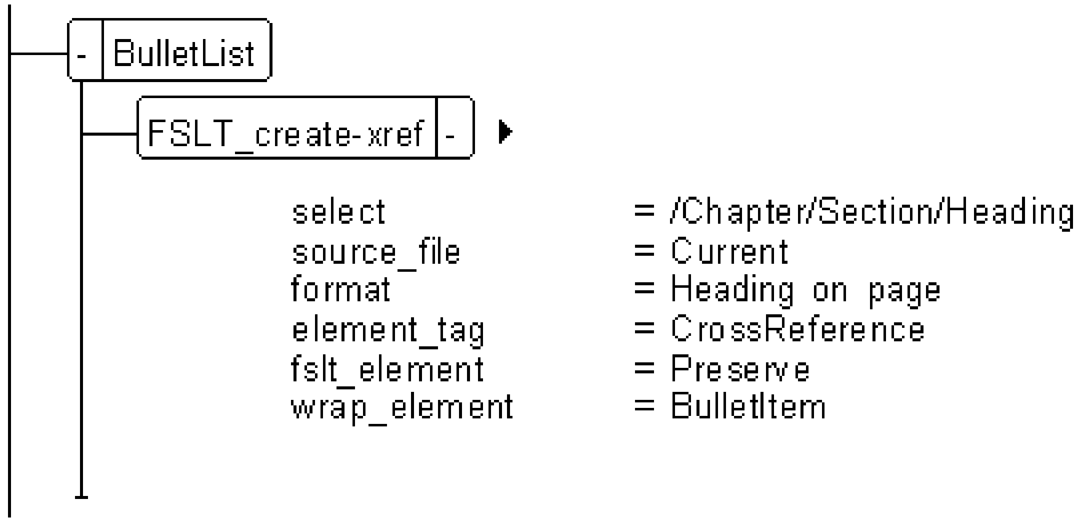
In addition, if a source `ID` attribute is found to be empty, FrameSLT generates a unique ID and populates the attribute, such that the cross-reference can be completed. Afterwards, you must save the document containing the source element with the new unique ID, otherwise *the cross-reference will become unresolved after you close the source file*.

FSLT_create-xref attributes

Attribute	Description
`select`	XPath expression for the query. Cross-references will be created for each match made.
`source_file`	Source file or book for the XPath query. For more information on source files, see *"Source file details"* on page 80.
`element_tag`	Element tag for the generated cross-references. This tag must be a valid tag for cross-references, according to the stylesheet's EDD.
`format`	Format for the cross-reference, from the stylesheet's template.
`wrap_element`	An optional element to wrap each generated cross-reference.
`fslt_element`	Whether or not to preserve the `FSLT_create-xref` element following a transformation. For more information, see *"About preserving transformation elements after a transformation"* on page 85.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the "current" stylesheet or document. For more information, see *"About starting contexts"* on page 82.

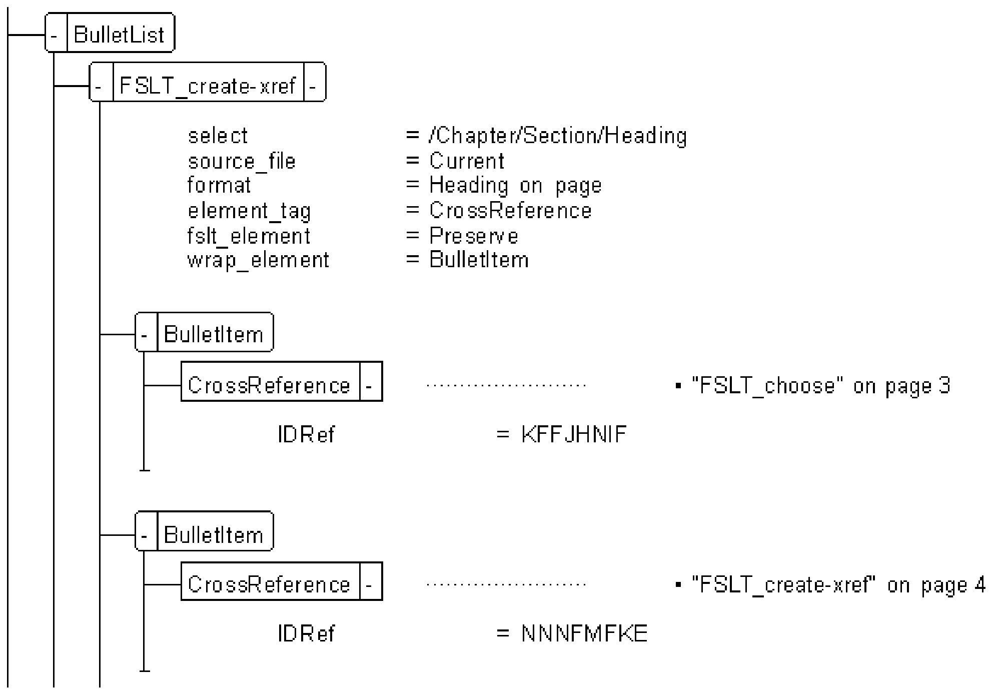
FSLT_create-xref example

The following figure shows the actual element structure used to create the cross-reference list at the beginning of this chapter:



```
select          = /Chapter/Section/Heading
source_file     = Current
format          = Heading on page
element_tag     = CrossReference
fslt_element    = Preserve
wrap_element    = BulletItem
```

In this case, the `FSLT_create-xref` element looked for all `Heading` elements that were a child of a `Section` element, which in turn was a child of the highest-level `Chapter` element. In other words, all first-level headings. For all matches made, it generated a cross-reference with the `CrossReference` element tag and the "Heading on page" format, and wrapped it in a `BulletItem` element. The resulting structure tree looked as follows:



```
- BulletList
  - FSLT_create-xref -
      select          = /Chapter/Section/Heading
      source_file     = Current
      format          = Heading on page
      element_tag     = CrossReference
      fslt_element    = Preserve
      wrap_element    = BulletItem
    - BulletItem
        CrossReference -  ........................  ▪ "FSLT_choose" on page 3
            IDRef         = KFFJHNIF
    - BulletItem
        CrossReference -  ........................  ▪ "FSLT_create-xref" on page 4
            IDRef         = NNNFMFKE
```

Because the element was set up to be preserved, the transformation could be run repeatedly as desired to update the cross-reference list.

FSLT_for-each

`FSLT_for-each` elements allow you to repeat a particular “template” for each match of a specified XPath expression. It is analogous to its XSLT counterpart, `for-each`.

FSLT_for-each processing

`FSLT_for-each` performs a normal XPath query, and for each match made, it adds its contents to the stylesheet. Sometimes called a “template,” the entire contents of the `FSLT_for-each` are added once for each match. In many ways, `FSLT_for-each` is similar to `FSLT_if`, except that `FSLT_if` stops at the first match and adds its contents one time only, while `FSLT_for-each` continues the process until all possible matches are exhausted, within the scope of the “max_matches” attribute.

On its own, `FSLT_for-each` does not retrieve any content from your source files. However, it can contain any number of other transformation elements that do, such as `FSLT_copy-of` and `FSLT_value-of`. As with all transformation elements, normal cascading context rules apply, such that the `FSLT_for-each` element will pass down the context of each match to its descendants before adding them to the stylesheet. For more information, see *“About cascading contexts”* on page 83.

Due to the templating concept and the structural changes caused by `FSLT_for-each`, the original transformation element and its contents are removed from the stylesheet after transformation. If the XPath makes no match at all, the transformed document will appear as if the element had simply been deleted.

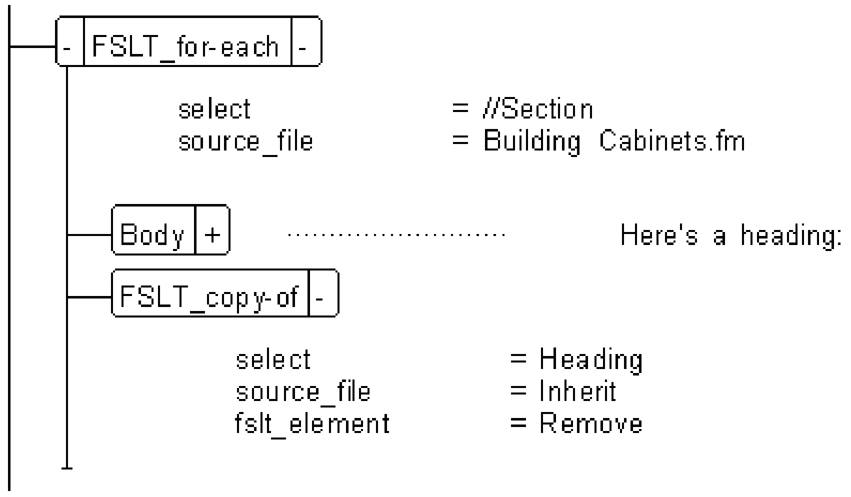
Tip: You can sort the content added to the stylesheet, either alphabetically or numerically, with an `FSLT_sort` element.

FSLT_for-each attributes

Attribute	Description
`select`	XPath expression for the query. For each match, the contents (or template) of the element are added to the stylesheet.
`source_file`	Source file or book for the XPath query. For more information on source files, see *“Source file details”* on page 80.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the “current” stylesheet or document. For more information, see *“About starting contexts”* on page 82.
`max_matches`	The maximum number of matches permitted for this element, with zero (0) indicating unlimited (match all).

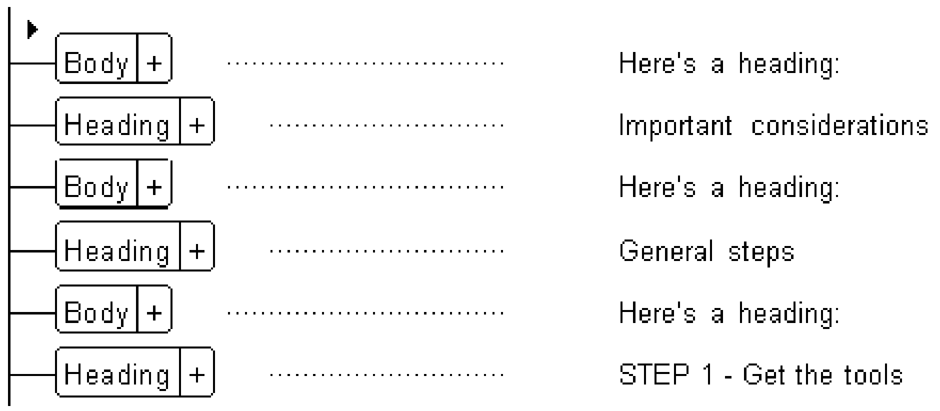
FSLT_for-each example

The following figure shows a sample `FSLT_for-each` element structure:



This `FSLT_for-each` element will be looking for all `Section` elements in `Building Cabinets.fm`. As each match is made, the `Body` and `FSLT_copy-of` elements are added to the stylesheet. During the process, the context of the matched `Section` element is passed down to the `FSLT_copy-of`, which is subsequently processed to copy over any `Heading` children of the `Section` element.

Assuming that each `Section` element in `Building Cabinets.fm` has exactly one child `Heading` element, the transformed structure might appear as follows:



Note that the original `FSLT_for-each` element and its contents are gone.

FSLT_if

An `FSLT_if` element allows you to perform an XPath-based evaluation, and if the evaluation holds true, the contents of the element are added to the stylesheet. Otherwise, the element and its contents are removed.

FSLT_if processing

`FSLT_if` performs an XPath query until a single match is made, or the source files are exhausted. If a match is made, `FSLT_if` adds its contents to the stylesheet and discontinues searching. If no match is made, the `FSLT_if` element is simply deleted. In other words, if the XPath finds anything at all, the "if" evaluation is considered "true," and no further searching is required.

`FSLT_if` supports basic parameter evaluation for the XPath, for example:

```
$MyParameter="ThisValue"
```

For any parameter evaluation that is more complex, such as with the use of functions, you should start the XPath with a "to-self" axis, then put the evaluation into a predicate. For example:

```
.[contains("$MyParameter", "ThisValue")]
```

On its own, `FSLT_if` does not retrieve any content from your source files. However, it can contain any number of other transformation elements that do, such as `FSLT_copy-of` and `FSLT_value-of`.

Due to the templating concept and the structural changes caused by `FSLT_if`, the original transformation element and its contents are removed from the stylesheet after transformation. If the XPath makes no match at all, the transformed document will appear as if the element had never existed.

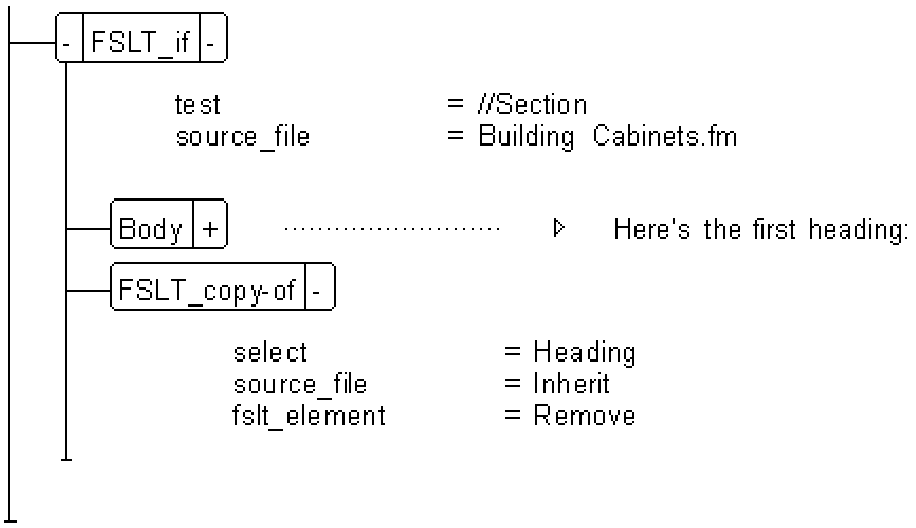
Note: In a normal XSLT environment, an `xsl:if` element itself should not pass any context to descendant transformation elements, unlike elements such as `xsl:for-each` which do pass down the context established by the XPath match from the `select` attribute. That is, if the XPath expression contained in the `test` attribute of `xsl:if` does make a match, it is for testing only and the context of the match does not get passed down. However, previous to version 2.0, FrameSLT erroneously did pass down the context from an `FSLT_if` XPath match. In an effort to fix this problem while maintaining backwards compatibility, the FrameSLT preferences now include an option to process in either fashion. For more information, see *"Preferences"* on page 10.

FSLT_if attributes

Attribute	Description
`test`	XPath expression to test for a match. If a match is made, the contents (or template) of the element are added to the stylesheet.
`source_file`	Source file or book for the XPath query. For more information on source files, see *“Source file details”* on page 80.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the “current” stylesheet or document. For more information, see *“About starting contexts”* on page 82.

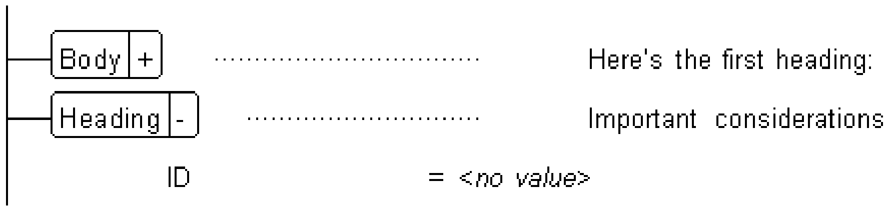
FSLT_if example

The following figure shows a sample `FSLT_if` element structure:



This `FSLT_if` element will be looking for a `Section` element in `Building Cabinets.fm`. When the first match is made, the `Body` and `FSLT_copy-of` elements are added to the stylesheet. During the process, the context of the matched `Section` element is passed down to the `FSLT_copy-of`, which is subsequently processed to copy over any `Heading` children of the `Section` element. After the first match of `FSLT_if`, its query stops and it is removed. The `FSLT_copy-of`, however, performs as normal, continuing its own query until all matches are exhausted.

In essence, this setup will copy over the first heading in `Building Cabinets.fm`. After transformation, the structure might appears as follows:



Note that the original `FSLT_if` element and its contents are gone.

FSLT_otherwise

The `FSLT_otherwise` element is an optional component of an `FSLT_choose` element structure. It contains the default content to be added to the stylesheet if no preceding `FSLT_when` elements make an XPath match.

FSLT_otherwise processing

See `FSLT_choose`.

FSLT_otherwise attributes

`FSLT_otherwise` has no attributes that you need to set. All XPath-based and other attributes are specified at the child `FSLT_when` element(s). `FSLT_otherwise` is only used as a default if all the XPath evaluations of the preceding `FSLT_when` elements fail to make an XPath match.

FSLT_otherwise example

See `FSLT_choose`.

FSLT_param

`FSLT_param` assigns a value to a parameter, either as retrieved from the contents of a matched node or as a static value. Once a parameter is defined, it can be retrieved for use in:

- XPath expressions (see *"About parameters in XPath expressions"* on page 85)
- Source file paths (see *"Use of parameters in source file paths"* on page 81)
- `FSLT_value-of` elements

Tip: If you are new to XSLT terminology, you can think of a parameter as a variable that can be set during transformation, then its value retrieved later in the situations listed above.

Note that an `FSLT_param` element is ignored during API-based transformations if the named parameter is already defined. For more information, see *"`SetParam`"* on page 139.

FSLT_param processing

`FSLT_param` operation is generally simple; however, the flexibility allowed with its configuration can cause confusion. When encountered, it assigns a value to the named parameter by either:

- Using the specified static value

  -or-
- Making an XPath query and retrieving the contents of the first matched node, up to the first 100 characters for element matches or the first value for attribute matches. If no match is made or the matched node is empty, the parameter is assigned an empty string.

For any transformation launched manually, no parameters are defined until `FSLT_param` elements are encountered. In other words, parameter assignment does not carry over to subsequent transformation actions. Also, unlike XSLT, parameters have no scope and may be redefined an indefinite number of times.

Note: An exception exists with API-launched transformations, which allow the predefinition of parameters. For more information, see *"`SetParam`"* on page 139.

`FSLT_param` attributes

Attribute	Description
`name`	Parameter name. The dollar sign ($) used when the parameter is referenced elsewhere is not required.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the “current” stylesheet or document. For more information, see *“About starting contexts”* on page 82.
`select`	Either: • A static value for the parameter enclosed in single quotes (') -or- • An XPath expression to query for the parameter value
`source_file`	Source file or book for the XPath query. If you are specifying a static value for the parameter, select the current document if no file context is available to be inherited. For more information on source files, see *“Source file details”* on page 80.
`fslt_element`	Whether or not to preserve the `FSLT_param` element following a transformation. For more information, see *“About preserving transformation elements after a transformation”* on page 85.

`FSLT_set-attribute`

`FSLT_set-attribute` allows you to set an attribute of a parent or preceding-sibling element. It can use XPath to find the value for the attribute, or you can specify a preset value. In either case, XPath is required, and a match must be made for the attribute to be set at all.

`FSLT_set-attribute` processing

`FSLT_set-attribute` performs an XPath query until a single match is made, or the source files are exhausted. If a match is made, `FSLT_set-attribute` sets the value of the specified attribute to either the contents of the matched node (element or attribute), or to a preset value. If no match is made, the attribute is not set at all.

Because an XPath match is required, if you are specifying a preset value that you want to be set every time, your XPath should be a generic expression that is guaranteed to make a match. For example, the expression `/*` used on the `Current` source file will match the highest-level element of the stylesheet, which by nature always exists.

`FSLT_set-attribute` can set an attribute on either a parent or preceding-sibling element. This specification is part of the required `FSLT_set-attribute` parameters. If the specified attribute cannot be found for any reason, a warning is printed to the error report and transformation continues.

`FSLT_set-attribute` adds the specified or derived value to any existing values of the attribute. It does not replace or delete any existing values. Also, if the XPath matches an attribute, only the first value of the attribute is extracted and applied to the new attribute.

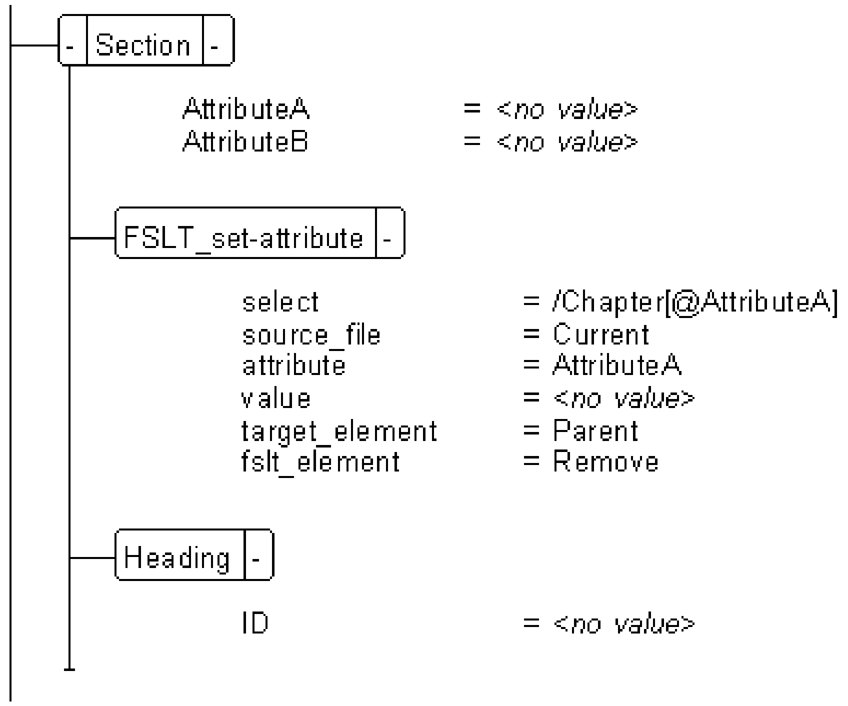
Tip: For wide-scale attribute manipulation, the FrameSLT Node Wizard may be more appropriate for some situations, used on the stylesheet after transformation. For more information, see *“Chapter 3 The Node Wizard and Other Utilities”* on page 35.

FSLT_set-attribute attributes

Attribute	Description
`select`	XPath expression for the query. If a match is made, the specified attribute is set, either with a preset value, or with the content derived from the matched node. If a preset value is specified, it always overrides the matched node content.
`source_file`	Source file or book for the XPath query. For more information on source files, see *“Source file details”* on page 80.
`attribute`	Name of the attribute to set. Case is considered.
`value`	Preset value, which if specified, is the value applied to the attribute. If specified, this value always overrides the content of the matched node. If the XPath makes no match, however, no value is applied at all.
`target_element`	The element containing the attribute to set, either a parent or preceding-sibling element, in relation to the `FSLT_set-attribute` element.
`fslt_element`	Whether or not to preserve the `FSLT_set-attribute` element following a transformation. For more information, see *“About preserving transformation elements after a transformation”* on page 85.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the “current” stylesheet or document. For more information, see *“About starting contexts”* on page 82.

FSLT_set-attribute example

The following figure shows a sample `FSLT_set-attribute` element:



```
- Section -
    AttributeA          = <no value>
    AttributeB          = <no value>

  FSLT_set-attribute -
    select              = /Chapter[@AttributeA]
    source_file         = Current
    attribute           = AttributeA
    value               = <no value>
    target_element      = Parent
    fslt_element        = Remove

  Heading -
    ID                  = <no value>
```

The `FSLT_set-attribute` element will attempt to find an `AttributeA` on the highest-level `Chapter` element of the current stylesheet. If found, it will extract the value of the `Chapter`'s `AttributeA` and apply it to the `Section`'s `AttributeA`. If the highest-level element is not named `Chapter`, or it has no `AttributeA`, nothing happens.

Note that the `value` attribute is unspecified. If specified, and the XPath made a match, this value would be applied to `AttributeA`, rather than the content derived from the XPath match.

FSLT_set-marker

`FSLT_set-marker` allows you to set the text of a parent or preceding-sibling marker element. It can use XPath to find the text for the marker, or you can specify a preset value. In either case, XPath is required, and a match must be made for the marker to be set at all.

FSLT_set-marker processing

`FSLT_set-marker` performs an XPath query until a single match is made, or the source files are exhausted. If a match is made, `FSLT_set-marker` sets the text of the specified marker to either the contents of the matched node (element or attribute), or to a preset value. If no match is made, the marker is not set at all.

Because an XPath match is required, if you are specifying a preset value that you want to be set every time, your XPath should be a generic expression that is guaranteed to make a match. For example, the expression `/*` used on the `Current` source file will match the highest-level element of the stylesheet, which by nature always exists.

`FSLT_set-marker` can set either a parent or preceding-sibling marker element. This specification is part of the required `FSLT_set-marker` parameters. Because marker elements generally do not allow children, normally the "preceding-sibling" specification should be used. If the specified marker element cannot be found for any reason, a warning is printed to the error report and transformation continues.

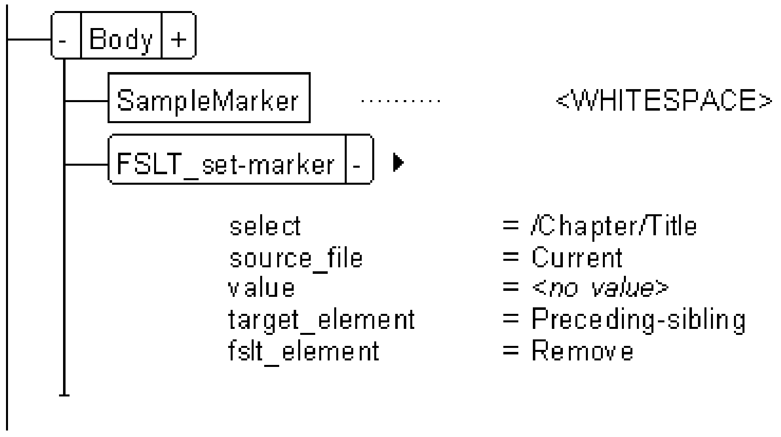
Note: `FSLT_set-marker` replaces any text currently assigned to the target marker.

FSLT_set-marker attributes

Attribute	Description
`select`	XPath expression for the query. If a match is made, the specified marker is set, either with a preset value, or with the content derived from the matched node. If a preset value is specified, it always overrides the matched node content.
`source_file`	Source file or book for the XPath query. For more information on source files, see *"Source file details"* on page 80.
`value`	Preset value, which if specified, is the text applied to the marker. If specified, this value always overrides the content of the matched node. If the XPath makes no match, however, no value is applied at all.
`target_element`	The element representing the marker to set, either a parent or preceding-sibling element, in relation to the `FSLT_set-marker` element.
`fslt_element`	Whether or not to preserve the `FSLT_set-marker` element following a transformation. For more information, see *"About preserving transformation elements after a transformation"* on page 85.
`starting_context`	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the "current" stylesheet or document. For more information, see *"About starting contexts"* on page 82.

FSLT_set-marker example

The following figure shows a sample `FSLT_set-marker` element:


```
- Body +
  SampleMarker   ..........   <WHITESPACE>
  FSLT_set-marker -
      select            = /Chapter/Title
      source_file       = Current
      value             = <no value>
      target_element    = Preceding-sibling
      fslt_element      = Remove
```

The `FSLT_set-marker` element will attempt to find a `Title` element child of the highest-level `Chapter` element of the current stylesheet. If found, it will extract the text of the `Title` apply it to the `SampleMarker` marker. If the highest-level element is not named `Chapter`, or it has no `Title` child.

Note that the `value` attribute is unspecified. If specified, and the XPath made a match, this value would be applied to `SampleMarker`, rather than the content derived from the XPath match.

FSLT_sort

FSLT_sort allows you to sort the content added to the stylesheet by an FSLT_for-each element, or table rows added by an FSLT_tablerow element, either alphabetically or numerically. The sort criterion is based on a text string or number derived with an XPath expression. This element is analogous to its XSLT counterpart, sort.

FSLT_sort processing

If present, FSLT_sort must be the first child of an FSLT_for-each or FSLT_tablerow element, where it is processed once for each match made by the respective transformation element. Each time, FSLT_sort performs an XPath query, normally starting from an inherited context, until a single match is made. Once a match is made, FrameSLT evaluates the content of the matched node (element or attribute), versus the contents of all previously matched nodes, and determines where to place the remaining contents of the FSLT_for-each element or the new table row. In other words, FSLT_for-each or FSLT_tablerow adds its content to the stylesheet once for each match, and FSLT_sort decides where to place that content in relation to previously-placed content. If FSLT_sort cannot make a match itself, it has no effect.

As mentioned previously, the criterion for evaluation is the content of a node matched with XPath. Nearly always, the XPath starts at a context set from the FSLT_for-each or FSLT_tablerow element match, evaluating some nearby element, or perhaps the element that FSLT_for-each or FSLT_tablerow matched itself. For example, if the XPath expression of the FSLT_sort element is a simple period (.), which means “go-to-self,” the sort evaluations are performed based on whichever node was matched by FSLT_for-each or FSLT_tablerow.

FSLT_sort allows the specification of either a numeric or string data type for the evaluation. In many cases, the sort results may be the same. However, some numbers, as a string, are lexically “smaller” than the equivalent numeric evaluation. For example, a text evaluation will regard “10” as smaller than “2,” while a numeric evaluation would return the opposite. Therefore, if you know for certain that all the evaluation items will be numbers, you should select the numeric comparison.

Before using FSLT_sort note the following:

- Currently, FrameSLT only allows a single FSLT_sort element per FSLT_for-each or FSLT_tablerow loop.
- Currently, numeric evaluations currently support integers only.
- FSLT_sort does not provide a source_file specification, because the only logical query would be on the same document queried by the FSLT_for-each or FSLT_tablerow element.

Special note on using FSLT_sort with FSLT_tablerow

When used to sort table rows generated by FSLT_tablerow, the results are conceptually similar as when used with FSLT_for-each. The rows of the resulting table are sorted according to the FSLT_sort XPath match, the same as they would be in any other circumstance.

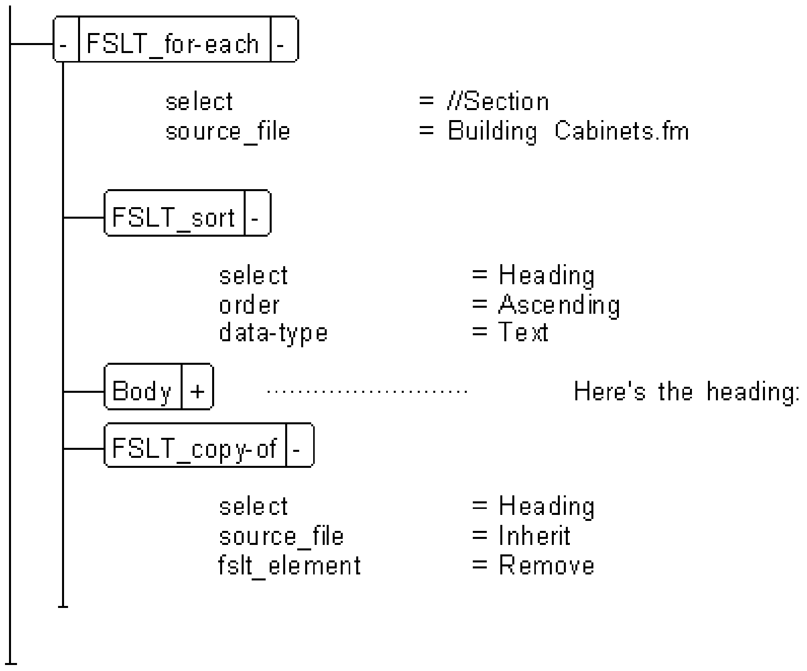
However, FSLT_tablerow/FSLT_sort combinations may require an additional consideration if you have multiple sibling FSLT_tablerow elements under one FSLT_table structure. You can have multiple FSLT_tablerow elements generating rows, each with its own FSLT_sort child, but this setup must be constructed carefully. If each FSLT_sort has the same order (ascending/descending) and type (text/numeric) settings, the sort should work as expected, and all generated rows will be sorted together. However, if you mix the order and/or type between FSLT_sort elements under a single FSLT_table structure, the sorting results may be unpredictable.

FSLT_sort attributes

Attribute	Description
`select`	XPath expression for the query. If a match is made, the specified marker is set, either with a preset value, or with the content derived from the matched node. If a preset value is specified, it always overrides the matched node content.
`order`	Order by which to sort, either ascending or descending.
`data-type`	Type of data to be evaluated. For more information, see *“FSLT_sort processing”* on page 104.

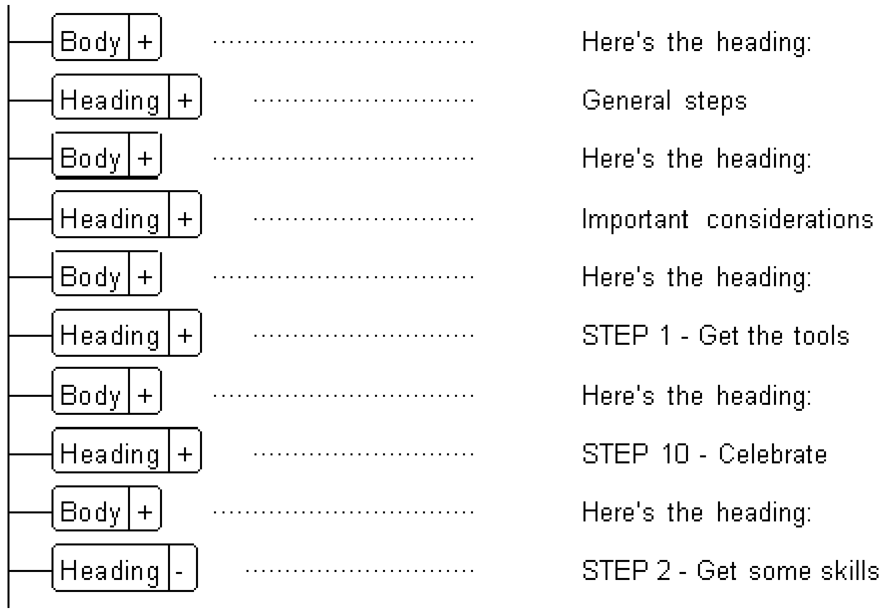
FSLT_sort example

The following figure shows a sample `FSLT_for-each` / `FSLT_sort` element setup:



The `FSLT_for-each` element queries `Building Cabinets.fm`, matching all `Section` elements. For each match, the contents of the `FSLT_for-each` element (excluding `FSLT_sort`) are added to the stylesheet. Before they are added, however, the `FSLT_sort` evaluates the text of the `Heading` child of the matched `Section` element and determines where to place the material. This process occurs once for each match made by `FSLT_for-each`.

After transformation, the element tree might appear as follows:



Note the alphabetical order of the text of the `Heading` elements. Also note how "STEP 10" was seen as lexically "smaller" than "STEP 2."

FSLT_table

`FSLT_table` allows you build structured FrameMaker tables with content extracted from your source files and/or other static content. `FSLT_table` and associated elements are required for table construction because of the specialized nature of FrameMaker table and table component elements. `FSLT_table` and associated elements are unique to FrameSLT and have no counterparts in XSLT.

FSLT_table structure and requirements

To use `FSLT_table`, you must essentially build a mock structure of the table you wish to generate, using the appropriate transformation elements for each required part. These transformation elements include:

- `FSLT_table`
- `FSLT_tabletitle`
- `FSLT_tableheading`
- `FSLT_tablebody`
- `FSLT_tablefooting`
- `FSLT_tablerow`
- `FSLT_tablecell`

Each of these elements represents a specific type of table component element that can be part of a FrameMaker table. A complete `FSLT_table` structure should resemble the basic structure of the intended output table in hierarchy and element order.

The requirements for using `FSLT_table` are stringent and must be followed carefully. All associated transformation elements must be in the correct position with valid parameters. During transformation, FrameSLT performs a comprehensive validation of all `FSLT_table` structures and will abort the process if any pieces are invalid or out of place. Although these rules place added responsibility on you as a stylesheet designer, they provide the distinct advantage of helping to ensure that your tables generate without error and look exactly as you had intended.

Tip: You can select **FrameSLT > Check Stylesheet** before launching a transformation to ensure that your tables are error-free.

Basic steps for creating a valid FSLT_table structure

Creating an `FSLT_table` structure is similar to creating a normal FrameMaker table structure, in that you must place the required component elements in the correct order and hierarchy to satisfy FrameMaker's table requirements. Like real tables, your `FSLT_table` structures must have certain elements, such as the table, body, a row, and at least one cell. Other transformation elements are optional, much like their FrameMaker counterparts, such as the heading and the footing.

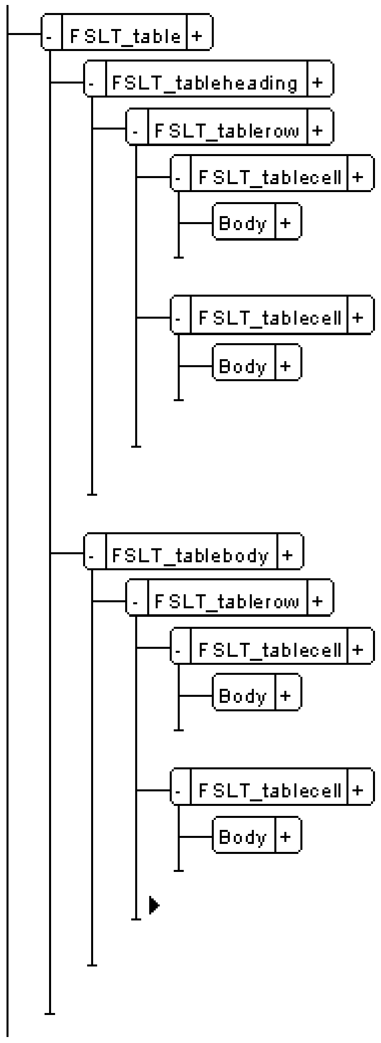
To create an `FSLT_table` structure, you might follow these general steps:

1. Insert an `FSLT_table` element, specifying the number of columns and other important settings.
2. Insert an `FSLT_tablebody` element, as a child of the `FSLT_table` element.
3. Insert an `FSLT_tablerow` element, as a child of the `FSLT_tablebody` element, specifying the XPath expression for row generation. For more information, see *"Generating rows with `FSLT_tablerow`"* on page 108.
4. Insert `FSLT_tablecell` elements under the `FSLT_tablerow` element. The number of cell elements must match the number of columns you specified at the `FSLT_table` element.

At this point, you have a valid `FSLT_table` structure, with all the basic requirements. You can then begin to add more elements as needed to complete the table, such as heading or footing elements, and contents for the table cells.

Complete FSLT_table structure example

The following figure illustrates a complete two-column `FSLT_table` structure, with an optional table heading and some contents within the cells. The attributes have been condensed for space considerations:



This table has two columns, as evidenced by the two `FSLT_tablecell` elements in each `FSLT_tablerow` element. Each `FSLT_tablecell` element contains a `Body` element, which may contain any text and is left as-is like any other non-transformation element. This table does not have a title or a footing, but if it did, those elements would be in the same positions that you would expect to see them in a normal FrameMaker table.

Generating rows with `FSLT_tablerow`

`FSLT_tablerow` is the primary generation element that causes your tables to grow. Each `FSLT_tablerow` element has an XPath expression, which performs normal queries of your specified source files. For each XPath match, a new row is generated, using the subordinate `FSLT_tablecell` elements as the template. You may have as many `FSLT_tablerow` elements as desired, with each being processed in order and generating rows as appropriate. All other transformation table elements contain no XPath, making them essentially static templates.

`FSLT_tablerow` elements initiate row generation only and do not extract any content from your source files. However, your `FSLT_tablecell` elements can contain any nature of valid transformation elements which may bring in content. Any time `FSLT_tablerow` makes a match and generates a row, the normal rules of context inheritance apply and subordinate transformation elements are passed the context of the XPath match. As always, though, you may choose whether or not to use that context, on an element-by element basis. For more information, see *“About cascading contexts”* on page 83.

Tip: If you know you want a particular row to be generated only once in all cases, your XPath should be a generic expression that is guaranteed to make a single match. For example, you may always want a single heading row, without any concern for the particulars of an XPath query. In this case, you could use the expression `/*` on the `Current` source file, which will always cause a single match of the highest-level element of the stylesheet.

Sorting generated table rows with FSLT_sort

Using an `FSLT_sort` element as the first child of an `FSLT_tablerow` element, you can sort the generated rows by an XPath-based alphabetical or numerical criterion. You can even sort rows generated by multiple sibling `FSLT_tablerow` elements. This functionality of `FSLT_sort` is unique to FrameSLT and has no counterpart in XSLT. For more information, see `FSLT_sort` and especially *"Special note on using FSLT_sort with FSLT_tablerow"* on page 104.

Other FSLT table component element setups

Table transformation elements other than `FSLT_table` and `FSLT_tablerow` simply require you to specify an element tag for their counterparts in the generated table. In most respects, they perform a simple templating function, laying out a precise table structure that FrameSLT can follow to generate the final table.

Checking an FSLT_table structure before transformation

Because of the strong possibility for errors, you should validate your stylesheets by selecting **FrameSLT > Check Stylesheet** before running transformations. This function performs the same pre-processing validation that occurs during transformation and will help you avoid aborts during actual transformations.

FSLT_table processing

Before transforming an `FSLT_table` structure, FrameSLT performs comprehensive validation that includes:

- Checking the required hierarchy and presence of transformation elements
- Verifying that specified element tags are valid for the components they will represent
- Ensuring that all `FSLT_tablerow` elements contain the same number of `FSLT_tablecell` children as there are columns specified at the `FSLT_table` element.

Afterwards, FrameSLT generates a table based on the template that the `FSLT_table` structure represents. Finally, FrameSLT uses the XPath to generate rows, based on settings at the `FSLT_tablerow` elements. For more information, see *"Generating rows with FSLT_tablerow"* on page 108.

For any heading or footing component whose `FSLT_tablerow` XPath expressions make no matches, no rows are generated, and therefore those components are removed from the final table. If the same situation occurs with the `FSLT_tablebody` element, the entire table is removed, because a FrameMaker table must have a body. No warning is given if table generation fails due to unsuccessful XPath queries.

FSLT_table attributes

Attribute	Description
`element_tag`	Valid table element tag from the stylesheet's EDD. An invalid tag will abort the transformation.
`format`	Valid table format from the stylesheet's template.

Attribute	Description
`num-columns`	Number of columns in the final table. Each `FSLT_tablerow` element in the template structure must contain this same number of `FSLT_tablecell` elements.
`col_widths`	Widths for each individual column, in inches.

FSLT_table example

To see a functional example of an `FSLT_table` element structure, see the `Sample7_Table.fm` file that came with FrameSLT.

FSLT_tablebody

The `FSLT_tablebody` element is a required component of an `FSLT_table` element structure. It acts as a simple template placeholder for the “real” table body element that will appear in the final table.

FSLT_tablebody processing

See `FSLT_table`.

FSLT_tablebody attributes

Attribute	Description
`element_tag`	A valid table body element tag from the stylesheet’s EDD.

FSLT_tablebody example

See `FSLT_table`.

FSLT_tablecell

The `FSLT_tablecell` element is a required component of an `FSLT_table` element structure. It acts as a simple template placeholder for a “real” table cell element that will appear in the final table.

FSLT_tablecell processing

See `FSLT_table`.

FSLT_tablecell attributes

Attribute	Description
`element_tag`	A valid table cell element tag from the stylesheet’s EDD.

FSLT_tablecell example

See `FSLT_table`.

FSLT_tableheading

The `FSLT_tableheading` element is an optional component of an `FSLT_table` element structure. It acts as a simple template placeholder for a “real” table heading element that will appear in the final table.

FSLT_tableheading processing

See `FSLT_table`.

FSLT_tableheading attributes

Attribute	Description
`element_tag`	A valid table heading element tag from the stylesheet’s EDD.

FSLT_tableheading example

See `FSLT_table`.

FSLT_tablefooting

The `FSLT_tablefooting` element is an optional component of an `FSLT_table` element structure. It acts as a simple template placeholder for a “real” table footing element that will appear in the final table.

FSLT_tablefooting processing

See `FSLT_table`.

FSLT_tablefooting attributes

Attribute	Description
`element_tag`	A valid table footing element tag from the stylesheet’s EDD.

FSLT_tablefooting example

See `FSLT_table`.

FSLT_tablerow

The `FSLT_tablerow` element is a required component of an `FSLT_table` element structure. It acts as a template placeholder for a “real” table row element that will appear in the final table, and it contains the XPath expression that contributes to the generation of table rows.

FSLT_tablerow processing

See `FSLT_table`.

FSLT_tablerow attributes

Attribute	Description
element_tag	A valid table row element tag from the stylesheet's EDD.
select	XPath expression for the query. For each match, a row is added to the table, using the template contained within the FSLT_tablerow element. The context of each match is passed down to any transformation elements contained within the FSLT_tablerow element. For more information, see *"Generating rows with FSLT_tablerow"* on page 108.
source_file	Source file or book for the XPath query. For more information on source files, see *"Source file details"* on page 80.
starting_context	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the "current" stylesheet or document. For more information, see *"About starting contexts"* on page 82.

FSLT_tablerow example

See FSLT_table.

FSLT_tabletitle

The FSLT_tabletitle element is an optional component of an FSLT_table element structure. It acts as a simple template placeholder for a "real" table title element that will appear in the final table.

FSLT_tabletitle processing

FSLT_tabletitle will cause a title to be added to the generated table. Because the title of a table is controlled by the table format, this element will cause a format override if the specified table format at the FSLT_table element does not included a title. If it does, the title will appear in the position indicated in the specified format. The final title will contain any content that the FSLT_tabletitle element contained.

For more processing information, see FSLT_table.

FSLT_tabletitle attributes

Attribute	Description
element_tag	A valid table title element tag from the stylesheet's EDD.

FSLT_tabletitle example

See FSLT_table.

FSLT_template

FSLT_template is a placeholder element for a "template" that is stored in another flow, or another document. A template can be any piece of structured content, including other transformation elements. During transformation, when FrameSLT encounters an

`FSLT_template` element, it finds the specified template and copies it into the stylesheet, and resumes transformation. Because `FSLT_template` elements can remain in a stylesheet document following transformation, they provide a means of transforming a stylesheet repeatedly using any nature of transformation setups, without having to create duplicate files to preserve the original stylesheet.

Note: `FSLT_template` has some loose similarities to its XSLT counterpart, `xsl:template`, but operates in a fundamentally different fashion. If you are familiar with XSLT, do not try to equate the two.

`FSLT_template` processing

In comparison to other transformation elements, `FSLT_template` processing is simple. When FrameSLT encounters this element during transformation, it looks for the corresponding template, identified by the `template_ID` attribute. If the corresponding template is found, the content is copied into the stylesheet as the contents of the original `FSLT_template` element.

When searching for the corresponding template, FrameSLT is actually searching for another `FSLT_template` element with the same specified ID. When found, FrameSLT copies the contents of the “source” `FSLT_template` element into the contents of the original `FSLT_template` element, and continues transformation.

`FSLT_template` does not use any XPath. It uses the template ID only to locate the source template.

Locations for “source” `FSLT_template` elements

When searching for a source `FSLT_template` element, FrameSLT looks in two places, in this order:

1 An “FSLT_template” flow in the specified source document, if it exists.

2 The main flow of the specified document.

Note: If the source document is specified as “Current,” that is, the stylesheet itself, FrameSLT looks in the “FSLT_template” flow only.

The source `FSLT_template` element can be anywhere in these locations, nested in any nature of structural organization. FrameSLT looks only for an element with the same `template_ID` attribute, and if found, copies the contents of it into the original `FSLT_template` in your stylesheet document.

Tip: For the source file, you can also specify a book. In this case, FrameSLT will step through the entire book looking for the corresponding source `FSLT_template` element. Note that for each chapter file, it will attempt to look in the “FSLT_template” flow first.

About the “FSLT_template” flow

As mentioned in previous sections, FrameSLT always searches in an “FSLT_template” flow first, if it exists. The primary intent of this functionality is to allow you to put templates on the reference pages of your stylesheet. That is, you can create a new reference page, create a flow on it called “FSLT_template,” and put all your templates there. In this manner, your templates always remain with your document.

You do not necessarily need to use an “FSLT_template” flow at all, if you want to store your templates in the main flow of a separate document. This functionality is provided simply as a convenience should you choose to use it.

To create an “FSLT_template” flow on your reference pages, follow these general steps:

1 Select **View > Reference Pages**.

2 Select **Special > New Reference Page**.

3 In the add page dialog, enter **FSLT_template**.

> ***Note:*** The name of the page actually doesn't matter, but it may help you keep your reference pages in order.

4 On the new reference page, which was probably added at the end of the pages, draw a new text frame.

5 Select the object pointer tool, right-click on the new frame, and select **Object Properties**.

6 Under **Flow Tag**, enter **FSLT_template**, and select **Autoconnect**.

> ***Note:*** The flow name must be absolutely correct, including case.

7 Begin a structure tree in the flow just like you would the main flow, and place the desired source `FSLT_template` elements anywhere you prefer.

Using `FSLT_template` to facilitate complex re-transformations

Normally, many transformation element types must be removed following a transformation, such as `FSLT_for-each` and `FSLT_table`. Therefore, to use these types of elements "as is," you must create a duplicate document when transforming if you want to preserve the original stylesheet.

However, because `FSLT_template` can remain in a stylesheet through repeated transformations, you can use it to retrieve complex transformation setups from elsewhere "on-the-fly." That is, it can act as a placeholder for a detailed transformation setup that gets copied in as original at the time of transformation, every transformation. This allows you to re-transform the same document repeatedly without any loss of integrity, regardless of the setup.

Note: During transformation, the first thing that FrameSLT does to `FSLT_template` is delete all its current contents. Then, it finds the corresponding source element and copies over the content. In this manner, the process is always a pure "refresh."

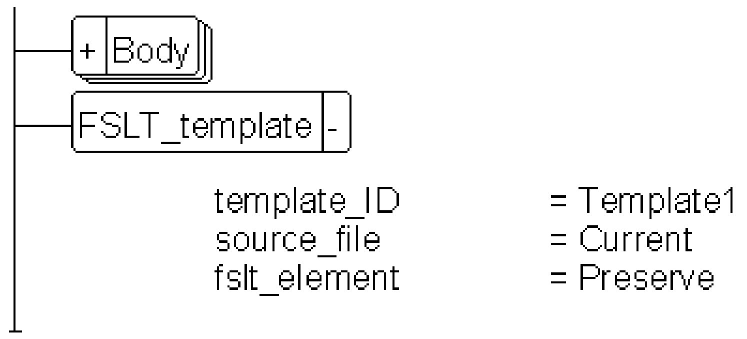
`FSLT_template` attributes

Note: When a source `FSLT_template` is located and the content is copied over, only the content gets copied. The source `FSLT_template` element itself is never copied. Therefore, the `source_file` and `fslt_element` attributes are irrelevant for source elements.

Attribute	Description
`template_ID`	ID for the template, which must be identical between the original and source `FSLT_template` elements. This value can be any alphanumeric string up to 255 characters.
`source_file`	Source file or book to search for the source `FSLT_template` element. This attribute is only relevant for "original" elements.
`fslt_element`	Whether or not to preserve the `FSLT_template` element following a transformation. This attribute is only relevant for "original" `FSLT_template` elements. For more information, see *"About preserving transformation elements after a transformation"* on page 85.

`FSLT_template` example

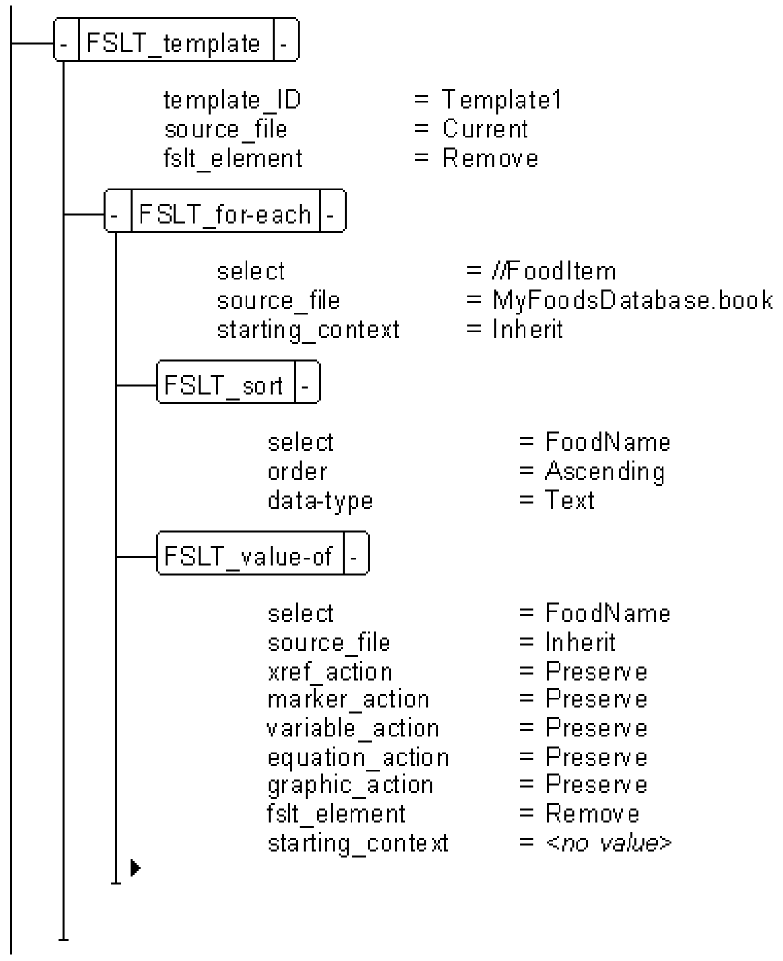
The following figure shows an "original" `FSLT_template` element in the main flow of a stylesheet, and afterwards the corresponding template element in the "FSLT_template" flow on a reference page:



```
+ Body
FSLT_template -

    template_ID        = Template1
    source_file        = Current
    fslt_element       = Preserve
```

FSLT_template element in the main flow

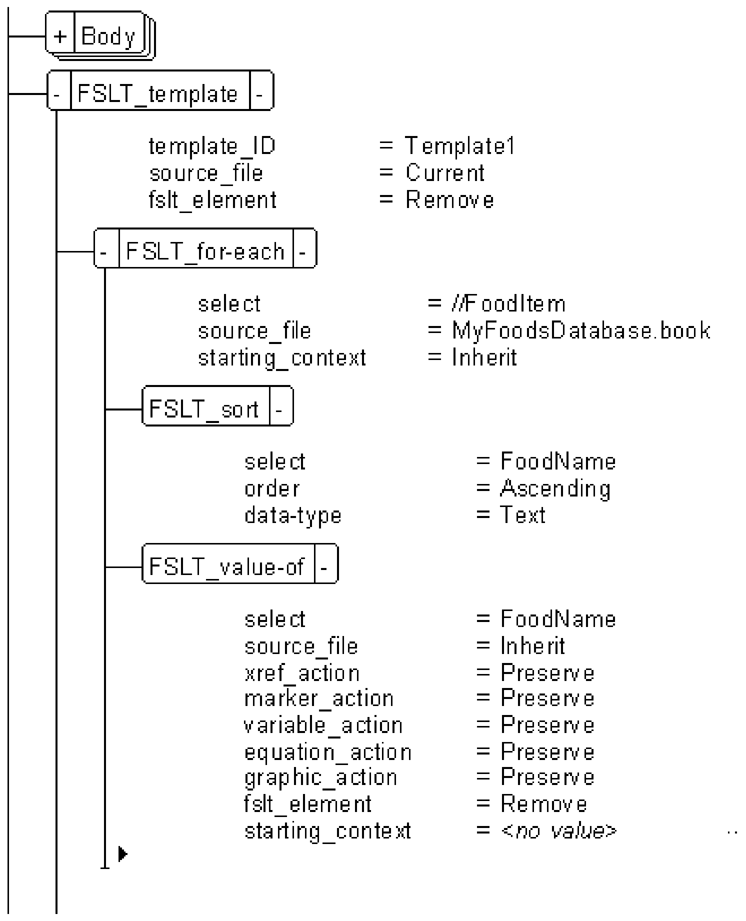


```
- FSLT_template -
    template_ID        = Template1
    source_file        = Current
    fslt_element       = Remove
  - FSLT_for-each -
      select             = //FoodItem
      source_file        = MyFoodsDatabase.book
      starting_context   = Inherit
    FSLT_sort -
        select             = FoodName
        order              = Ascending
        data-type          = Text
    FSLT_value-of -
        select             = FoodName
        source_file        = Inherit
        xref_action        = Preserve
        marker_action      = Preserve
        variable_action    = Preserve
        equation_action    = Preserve
        graphic_action     = Preserve
        fslt_element       = Remove
        starting_context   = <no value>
```

FSLT_template element in the "FSLT_template" flow, on a reference page

Note that the `template_ID` attributes are the same. During transformation, when the original `FSLT_template` element is encountered, all its contents are deleted, and the contents of the source element are copied in. For this example, the original `FSLT_template` element has no contents, so nothing needs to be deleted.

In the moment after these two steps are completed, the setup in the original `FSLT_template` element would look as follows:



```
+ Body
- FSLT_template -
    template_ID         = Template1
    source_file         = Current
    fslt_element        = Remove
  - FSLT_for-each -
      select              = //FoodItem
      source_file         = MyFoodsDatabase.book
      starting_context    = Inherit
    FSLT_sort -
        select              = FoodName
        order               = Ascending
        data-type           = Text
    FSLT_value-of -
        select              = FoodName
        source_file         = Inherit
        xref_action         = Preserve
        marker_action       = Preserve
        variable_action     = Preserve
        equation_action     = Preserve
        graphic_action      = Preserve
        fslt_element        = Remove
        starting_context    = <no value>
```

Setup the moment after processing the FSLT_template element

Note that the setup pictured above is only momentary, and you should never actually see it. FrameSLT should continue transforming, starting with the content it just copied in, including the `FSLT_for-each`, `FSLT_sort`, and `FSLT_value-of` elements.

FSLT_value-of

`FSLT_value-of` is one of the primary elements for retrieving content from your source files. It performs an XPath query and copies over the contents of any element or attribute that it matches, normally discarding all element tags of any matched and subordinate elements, as applicable. At a fundamental level, `FSLT_value-of` is analogous to its XSLT counterpart `value-of`, except that it also allows special provisions for special FrameMaker element types.

FSLT_value-of processing

`FSLT_value-of` queries your source files based on the specified XPath, and for any element that it matches, it copies the contents of it to the stylesheet. For “normal” elements, it does not copy the matched element tag itself, and normally does not copy any descendant element tags. It does, however, retain all subordinate content, essentially merging it all together as applicable. Because of this functionality, `FSLT_value-of` is frequently used to extract content from a source file without the element definition(s), for the purpose of retagging it once in the stylesheet.
`FSLT_value-of` can also match and retrieve values from attributes. If a matched attribute has multiple values, only the first value is retrieved.

Tip: FSLT_value-of can also emit the value of a previously-defined parameter. For more information, see *"FSLT_value-of attributes"* on page 117.

FSLT_value-of includes several options for handling special FrameMaker element types, which is unique to FrameSLT. Certain element types, including graphics, cross-references, markers, variables, and equations, are essentially "empty" element tags with the respective FrameMaker object behind them. If their element tags were removed, they would likewise be removed from the document. Therefore, on a case-by-case basis, you can select how to handle these element types, either to preserve, remove, or in some cases, convert to text. These settings apply to the element matched by the XPath, and any descendant elements. In XSLT, if a value-of element acted on any of these special element types in their XML formats, you would always lose them during the content retrieval process.

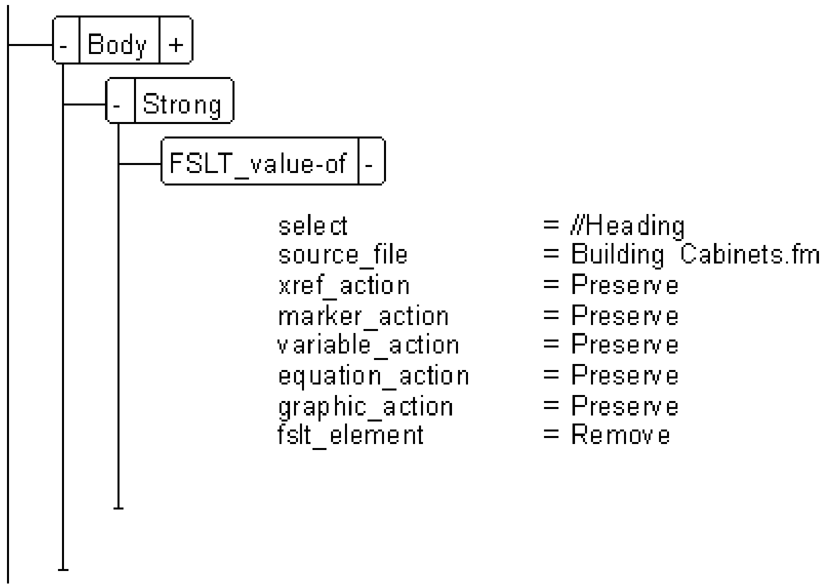
FSLT_value-of continues matching and extracting content until all XPath matches have been exhausted. If your XPath matches multiple elements with paragraph content, your resulting content in the stylesheet is likely to contain multiple paragraphs wrapped within a single element, which is normally not recommended.

FSLT_value-of attributes

Attribute	Description
select	XPath expression for the query. The contents of any element or attribute matched will be copied to the stylesheet. ***Note:*** You may also specify a parameter name, preceded by a dollar sign ($). If the parameter is defined, the value is copied to the stylesheet. If it is not defined, the string <UNDEFINED PARAMETER> is printed instead and the transformation continues. To be defined, a parameter must be defined by an FSLT_param element previously in the transformation process.
source_file	Source file or book for the XPath query. For more information on source files, see *"Source file details"* on page 80.
xref_action marker_action variable_action equation_action graphic_action	Individual settings for handling special FrameMaker element types within any content copied over by FSLT_value-of. For more information, see *"FSLT_value-of processing"* on page 116.
fslt_element	Whether or not to preserve the FSLT_value-of element following a transformation. For more information, see *"About preserving transformation elements after a transformation"* on page 85.
starting_context	The starting context for the XPath query, either as inherited, implied by the XPath, or the transformation element itself. This option is only available if the element is querying the "current" stylesheet or document. For more information, see *"About starting contexts"* on page 82.

FSLT_value-of example

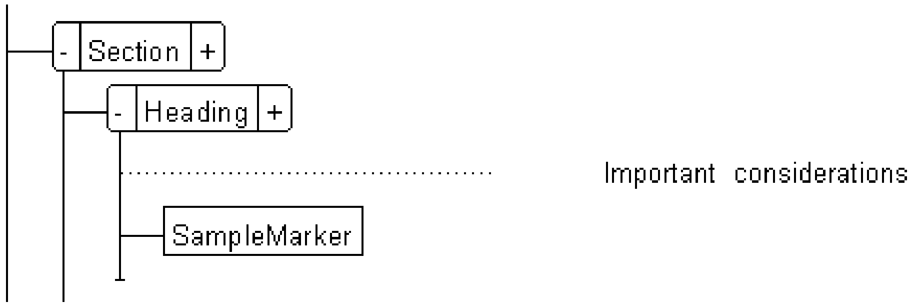
The following figure shows an FSLT_value-of element configured to match Heading elements in Building Cabinets.fm:



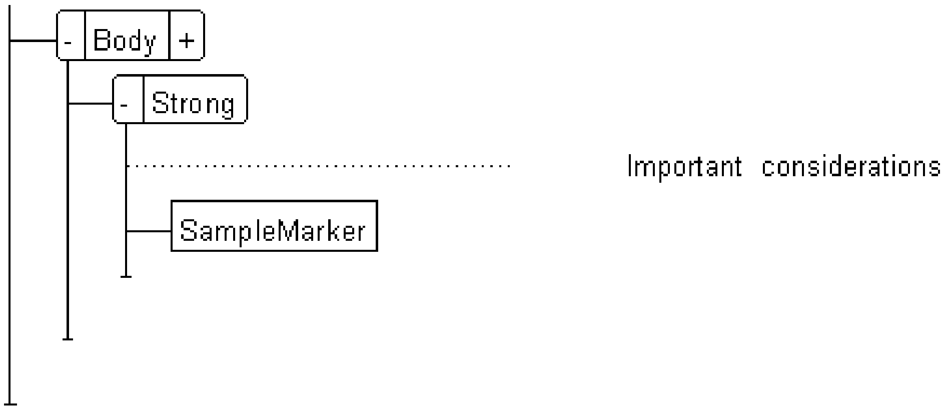
select	= //Heading
source_file	= Building Cabinets.fm
xref_action	= Preserve
marker_action	= Preserve
variable_action	= Preserve
equation_action	= Preserve
graphic_action	= Preserve
fslt_element	= Remove

This `FSLT_value-of` element is set to preserve all special element types within the retrieved content. Therefore, if the matched `Heading` elements contain any markers, graphics, or other special elements, they will be preserved in the final output with their original element definitions.

As an example, consider the following `Heading` element, with a `SampleMarker` marker element as a child:



If the `FSLT_value-of` element in the previous figure matched this element, the following would be the results:



Note how the content from the `Heading` element is placed where the `FSLT_value-of` element used to be. The `Heading` element tag has been removed as normal, but the `SampleMarker` tag and the associated marker has been preserved.

FSLT_when

The `FSLT_when` element is a required component of an `FSLT_choose` element structure. It contains the XPath that performs the conditional evaluations and determines which content, if any is added to the stylesheet.

`FSLT_when` supports basic parameter evaluation for the XPath, for example:

```
$MyParameter="ThisValue"
```

For any parameter evaluation that is more complex, such as with the use of functions, you should start the XPath with a “to-self” axis, then put the evaluation into a predicate. For example:

```
.[contains("$MyParameter", "ThisValue")]
```

FSLT_when processing

See `FSLT_choose`.

FSLT_when attributes

Attribute	Description
`test`	XPath expression to test for a match. The contents of any element or attribute matched will be copied to the stylesheet.
`source_file`	Source file or book for the XPath query. For more information on source files, see *“Source file details”* on page 80.

FSLT_when example

See `FSLT_choose`.

Chapter 6
External Calls to FrameSLT

Like many FrameMaker plugins, you can make external calls to FrameSLT to invoke XPath related functions, and use the results to perform customized actions within FrameMaker. Specifically, you can call FrameSLT to:

- Parse an XPath expression and find applicable nodes
- Allocate and deallocate memory associated with parsed XPath expressions

The exposure of these functions through the FrameMaker API essentially transforms FrameSLT into an XPath-based query engine that you can call for any nature of content management functions feasible within FrameMaker. For example, you could:

- Create a customized system of text insets or other content reuse
- Create a custom plugin that performs structure alterations after an XML import, without having to add complex code to an import/export client
- Create an automated system of assigning condition tags to elements based on attributes or element names

One of the keys to content management is being able to locate the content in question. Because XPath allows you to find very specific node instances, FrameSLT XPath opens the door to powerful content management, limited only by your imagination and end goals.

How to send an external call to FrameSLT

To call FrameSLT, you can use one of three methods:

- **With the FDK F_ApiCallClient() function, from another API client** If you are working on another FDK client, you can use `F_ApiCallClient()` to call FrameSLT. This function is part of the normal FDK library and does not require any changes to your normal project settings. For more information on the function itself, see the *FDK Developer's Reference* provided by Adobe with the FDK.
- **With ExtendScript (FrameMaker 10 and later)** The ExtendScript `CallClient()` function behaves identically to an API call with `F_ApiCallClient()`.
- **With FrameScript** FrameScript®, a scripting tool by Finite Matters, Ltd®, has a comparable function for calling FDK clients, `CallClient`. When called from FrameScript, FrameSLT behaves identically to a regular API call.
- **With FrameAC** FrameAC by Mekon® (www.mekon.com) is a plugin that enables developers to use Visual Basic to control FrameMaker. FrameAC also provides the ability to script calls to other API clients.

For any supported operation, you pass a string to FrameSLT which contains a command and any applicable parameters, and FrameSLT sends back a numeric code indicating the results. The syntax of these strings is the same for either API or scripting calls, and is explained in detail in this document.

Tip: The call descriptions and examples in this document are written from an FDK/API perspective, using `F_ApiCallClient()`. If you are using FrameScript or FrameAC, the basic call syntax will be the same, sent using the mechanism supported by the respective tool.

General information on external calls

Before you attempt to call FrameSLT, note the following:

- Calls and returns sometimes involve document and element IDs, instead of names. For example, when call FrameSLT to find an element node with XPath, it will return the ID of the element it finds. Therefore, to use external calls effectively, you must be familiar with element and document IDs and how to convert them into the desired results.
- The default delimiter string between arguments in a call to FrameSLT is three dashes (---). This delimiter can be changed with a `ChangeCallDelimiter` call.
- Due to the nature of `F_ApiCallClient()`, FrameSLT can only return a single integer after a call. No strings or other values can be returned. Therefore, all returns are in integer format and may represent items such as sequence numbers, element IDs, and error codes.
- Several calls to FrameSLT return zero (0) to indicate success, consistent with the behavior of other FDK functions. However, F_ApiCallClient() also returns zero if it fails to communicate at all with the specified API client. If you aren't sure whether your calls are reaching FrameSLT, you can call the special `Hello` command to verify that communications are getting through.
- With the exception of XPath expressions, call strings are generally not case-sensitive. For example, to parse an XPath string, you can send any case variation of the `ParseXPath` command name, such as `PARSEXPATH` or `parsexpath`.

Typical sequences of events

If you want to use XPath to navigate a document, you might:

1. First call `ParseXPath` to parse the expression(s) and retrieve internal sequence number(s)
2. Call `FindNextNode` to perform the navigation, sending it the sequence number you retrieved from `ParseXPath`.
3. After each XPath match (`FindNextNode` call), you could call `RetrieveAttrMatch` to retrieve the index of the matched attribute, if your expression matches attributes.
4. Once a query has exhausted all matches, you could reset the sequence with `ResetSequence` and start the query again, perhaps with a different context node or document.

If you want to run a Node Wizard script, you can simply call `RunNWScript` with the correct parameters. No preliminary steps with FrameSLT are necessary, unless you want to preconfigure one or more parameters with `SetScriptParm`.

If you want to transform a file, you can simply call `TransformFile` with the correct parameters. No preliminary steps with FrameSLT are necessary.

Call reference

This section details the external calls you can make to FrameSLT.

AllocateNodeHandlers

Clears the space used to hold parsed XPath data.

Syntax

```
F_ApiCallClient("FrameSLT", "AllocateNodeHandlers");
```

Usage description

This call clears and allocates the memory space used to hold parsed XPath data. No deallocation call is required beforehand. All parsed XPath data will be deleted, and any sequence numbers retrieved by previous `ParseXPath` calls will be rendered invalid.

Note: In previous versions of FrameSLT, this call was required before you could parse XPath. This is no longer true. All memory management is handled internally and you do not ever need to call `AllocateNodeHandlers`. It is maintained in the current version for general purpose and backwards compatibility only.

Returns

`F_ApiCallClient()` returns one of the following values after a `AllocateNodeHandlers` call:

Value	Meaning
0	Allocation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command

ChangeCallDelimiter

Changes the delimiter for external call arguments. The default upon startup is three dashed ("`---`").

Syntax

```
    F_ApiCallClient("FrameSLT", "ChangeCallDelimiterNewDelimiter");
```

Note: The new delimiter directly follows the `ChangeCallDelimiter` command. Do not separate them with the old delimiter. Anything following the command will be considered the new delimiter.

Returns

`F_ApiCallClient()` returns one of the following values:

Value	Meaning
0	Delimiter successfully changed. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	Unrecognized command. Make sure you spelled "`ChangeCallDelimiter`" correctly.
2	Incorrect number of arguments in the call string. Make sure you provided a new delimiter after `ChangeCallDelimiter`.

`ChangeCallDelimiter` syntax example

```
    F_ApiCallClient("InsetPlus", "ChangeCallDelimiter++++");
```

ClearScriptParms

Clears all parameter data currently in memory, whether set by a script or with `SetScriptParm`.

Syntax

```
F_ApiCallClient("FrameSLT", "ClearScriptParms");
```

Returns

`F_ApiCallClient()` returns one of the following values after a `ClearScriptParms` call:

Value	Meaning
0	Call was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.

DeallocateNodeHandlers

Clears the space used to hold parsed XPath data. This call performs the same operation as `AllocateNodeHandlers`.

Syntax

```
F_ApiCallClient("FrameSLT", "DeallocateNodeHandlers");
```

Usage description

This call clears and allocates the memory space used to hold parsed XPath data. No allocation call is required beforehand. All parsed XPath data will be deleted, and any sequence numbers retrieved by previous `ParseXPath` calls will be rendered invalid.

Note: In previous versions of FrameSLT, this call was recommended for cleanup following XPath usage, because memory handling required more management steps. This is no longer true. All memory management is handled internally and you do not ever need to call `DeallocateNodeHandlers`. It is maintained in the current version for general purpose and backwards compatibility only.

Returns

`F_ApiCallClient()` returns one of the following values after a `DeallocateNodeHandlers` call:

Value	Meaning
0	Deallocation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.

FindNextNode

Finds element nodes based on parsed XPath data, using a sequence number returned by `ParseXPath`.

Syntax

```
F_ApiCallClient("FrameSLT",
   "FindNextNode---SeqNumber---Document---ContextElemId---Flow");
```

where:

`SeqNumber`	Sequence number returned by `ParseXPath`, representing the desired parsed XPath expression.
`Document`	One of the following: • A document object handle in integer form (integer form of the FDK F_ObjHandleT type) • A fully-qualified path name of an open document In either case, the document to be queried must be currently open.
`ContextElemId`	The object handle of the context element, in integer form (integer form of the FDK F_ObjHandleT type). For more information on this parameter, see *"Usage description"* on page 125.
`Flow`	One of the following: • A flow object handle in integer form (integer form of the FDK F_ObjHandleT type) • A flow name, case-sensitive To indicate the main flow, you may also send "0" or "Main". If this argument is not sent at all, FrameSLT will assume the main flow. ***Note:*** The flow ID/name is only required if you send zero (0) for the `ContextElemId`. If you send a valid context element ID, you may send zero for the flow, or omit the argument entirely. For more information, see *"Usage description"* on page 125.

Usage description

`FindNextNode` uses parsed XPath data to find an element or attribute node, within a FrameMaker document you specify. The parsed XPath is identified by a sequence number returned by `ParseXPath`, so you must call `ParseXPath` to parse the XPath expression before you can use `FindNextNode`.

`FindNextNode` requires you to send the ID of a context element, which indicates the starting point for the query. XPath works in a sequential fashion, beginning at some established place within the structure tree and matching nodes until it runs out of matches. The matching is always based on some evaluation of context, which is true even for the first match.

If your XPath expression begins with a forward slash (that is, a go-to-root axis), the context element ID sent is actually ignored, because the first axis forces the context to the root. For example, with the following expression:

```
//Body
```

...the first axis will force the query to begin at the structural root, after which it will match any `Body` elements that are descendants of the highest-level element. With an expression such as this, the context element ID is irrelevant, and you can simply send a zero (0).

Conversely, if the XPath does not force the query to start at the root, you must send a context element ID that represents the starting point. For example, the following expression:

```
Body
```

...will match any `Body` elements that are children of the starting context element, whatever that may be. Therefore, this type of expression requires you to send the ID of that element from which the query should begin.

Note: Even when required, the context element ID is only used for the first match, because all subsequent matches either remember the original context or use a new context as established by the query itself. However, you should send the original context element ID with each `FindNextNode` call, otherwise FrameSLT may assume an error has occurred and return zero. This is true even if you sent originally sent a zero (0). If so, all subsequent `FindNextNode` should send zero as well.

You can call `FindNextNode` on any currently-parsed XPath expression, provided that you have a valid sequence number. Each sequence keeps track of its own query, and will always pick up where it left off with each subsequent `FindNextNode` call. You do not need to perform any steps to manage individual XPath queries, other than to simply call `FindNextNode`.

As mentioned earlier, each XPath query begins at some established starting point and matches nodes until there are none left to match. At the end or at any point in between, you can call `ResetSequence` to clear the internal sequence and start the query anew. After a `ResetSequence` call, you can send a new context element ID with `FindNextNode`, as applicable to the respective XPath expression.

Tip: A repeat call of `ParseXPath` on an already-parsed expression will also reset the respective sequence.

The flow name/ID is only required if you do not send a context element ID. If you do send a context element ID, the flow to query will be derived from that element, and you should simply send zero for the flow. Conversely, if you send zero for the context element ID, a flow ID or name is required such that FrameSLT knows which structure tree to query. Keep in mind that any FrameMaker document flow can be structured, and FrameSLT supports XPath queries on any structured flow.

Returns

`F_ApiCallClient()` returns one of the following values after a `FindNextNode` call:

Value	Meaning
0	No nodes found. When zero is returned, the sequence has been exhausted and all nodes have been located by previous `FindNextNode` calls. To start the sequence over, you can call `ResetSequence`. Without resetting the sequence, `FindNextNode` will return zero for any future calls. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string
2	Incorrect number of arguments sent with the command
4	Bad sequence number. Check to make sure you have sent the sequence number returned by `ParseXPath`.
6	Bad document argument. An invalid document ID or filename was sent. Make sure the argument represents the ID or filename of a valid, open document.

Value	**Meaning**
7	Bad flow argument. An invalid flow name or ID was sent.

Any integer greater than 100	An integer form of a matched element ID. The ID may be in one of two formats, according to a parameter set with `SetAppParm`:
	• An integer form of an F_ObjHandleT object handle (default). When calling FrameSLT from another FDK client, this type of ID is normally the most convenient, as it can be readily cast back to a valid object handle.
	• An element unique ID (FP_Unique) property. When calling FrameSLT from ExtendScript, this type of ID is normally the most convenient, as it is cumbersome to convert an FDK F_ObjHandleT ID into an ES object. With a unique ID, you can use the `GetUniqueObject()` method to convert to an ES object, for example:
	`var doc = app.ActiveDoc;`
	`var seqnum = CallClient("FrameSLT", "ParseXPath---//Body---False");`
	`CallClient("FrameSLT" , "SetAppParm---EC_ReturnIDType---UID");`
	`var id = CallClient("FrameSLT" , "FindNextNode---" + seqnum + "---" + doc.id + "---0---Main");`
	`var elem = doc.GetUniqueObject(Constants.FO_Element, id);`
	`alert(elem.ElementDef.Name);`
	Note: If your XPath expression matches attribute nodes instead of element nodes, the return will still be an element ID, representing the ID of the parent element. If you would like to retrieve the index of the matched attribute nodes, you can call `RetrieveAttrMatch` after `FindNextNode`.

Syntax examples

```
F_ApiCallClient("FrameSLT",
  "FindNextNode---21---67108880---0”);
F_ApiCallClient("FrameSLT",
  "FindNextNode---21---67108880---0---503586820");
F_ApiCallClient("FrameSLT",
  "FindNextNode---21---C:\MyDocs\Myfile.fm---0---Main");
```

Code sample

The following example shows the basic syntax of an actual `FindNextNode` call.

Note: Because the parsed XPath expression begins with a “go-to-root” axis, the context element is not important. Otherwise, you would need to have that ID as well.

```
. . .
F_ObjHandleT docId, elemId;
UCharT arg[50];
UIntT sequenceNumber;
IntT returnVal;

. . .
/* Parse the XPath */
sequenceNumber =
  F_ApiCallClient("FrameSLT", "ParseXPath---//Section/Body[1]---True");
```

```
/* Get a document ID */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Form the argument for the FindNextNode call */
F_Sprintf(arg, "FindNextNode---%d---%d---0---Main", sequenceNumber, docId);

/* Call FrameSLT to find the next node */
returnVal = F_ApiCallClient("FrameSLT", (StringT)arg);

/* Convert the returned integer ID back to an object handle */
elemId = (F_ObjHandleT)returnVal;

/* Report */
if(elemId > 20)
  F_ApiAlert("Found an element.", FF_ALERT_CONTINUE_WARN);
else if(elemId > 0)
  F_ApiAlert("An error occurred.", FF_ALERT_CONTINUE_WARN);
else
  F_ApiAlert("Nothing found. Sequence is spent.", FF_ALERT_CONTINUE_WARN);
```

GetAppParm

Retrieves a general application parameter.

Syntax

`F_ApiCallClient("FrameSLT", "GetAppParm---`*`Name`*`")`

...where the following table describes valid parameter names:

Name	**Description**
`EC_NWScriptsDoc`	Retrieves the F_ObjHandleT object ID of the currently-active Node Wizard Scripts document.
`EC_FrameSLTVersionMajor`	Retrieves the major version number of the plugin; for example, the “3” in v3.17. The value returned is actually the version number plus 100, so you should subtract 100 from the result to get the actual version number.
`EC_FrameSLTVersionMinor`	Retrieves the minor version number of the plugin; for example, the “17” in v3.17. The value returned is actually the version number plus 100, so you should subtract 100 from the result to get the actual version number.

Returns

`F_ApiCallClient()` returns one of the following values after a `SetScriptParm` call:

Value	**Meaning**
0	Operation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn’t work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.

Value	Meaning
`2`	Incorrect number of arguments sent with the command.
`5`	No active Node Wizard Scripts document is set or the active document is not currently open.
`20`	Invalid parameter name.
Any other value over 100	The requested parameter value.

Syntax example

```
returnVal =
  F_ApiCallClient("FrameSLT", "GetAppParm---EC_FrameSLTVersionMajor");
```

GetScriptParmByte

Gets an integer value representing a byte of currently-defined Node Wizard Script parameter value. It is intended as a means for retrieving a parameter value currently in memory, using a byte-by-byte retrieval and reconstruction methodology.

Note: If you are using ExtendScript, you may consider using an external object instead, which can return a string directly. The reconstruction of a string byte-by-byte can be difficult in a script, especially if multibyte characters are included. For more information, see *"Using FrameSLT as an ExtendScript external object"* on page 149.

Syntax

`F_ApiCallClient("FrameSLT", "GetScriptParmByte---`*`Name`*`")`

where:

`Name`	Parameter name. The preceding dollar sign ($) sometimes used for parameter notation is not required.

Usage description

`GetScriptParmByte` is provided to retrieve the value of a currently-defined script parameter, using a byte-by-byte methodology that can be used iteratively to reconstruct a string. The byte-by-byte method is required to overcome the `F_ApiCallClient()` limitation of returning a single integer only.

To allow for error-related returns in the lower integers, all byte values are returned plus 100. Therefore, you should subtract 100 from every byte value returned before adding it to your result string.

The first call to a given parameter retrieves the first byte, the second retrieves the second, and so forth. When the end of the string is reached, the call returns 100 (0) one time, then subsequent calls will resume at the beginning of the string. Note the following:

- You can use this call to retrieve the value of any parameter after a script run, whether the parameter was set by the script or previously with `SetScriptParm`.
- You should always iterate over the value of a single parameter entirely before attempting to retrieve the value of a second. If you attempt to retrieve bytes from a second parameter before completing the first, the results will be unpredictable.
- Because this methodology uses bytes and not code points, it is straightforward to reconstruct strings with any character encoding.

Returns

`F_ApiCallClient()` returns one of the following values after a `GetScriptParmByte` call:

Value	Meaning
0	A communication error occurred. Consider calling `Hello` to ensure connectivity with FrameSLT.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command
20	Invalid parameter name; that is, the parameter is not currently defined.
100	The end of the value string was reached.
A value between `101` and `355`	A byte value plus 100.

Syntax example

The following sample retrieves the value of the parameter “MyParameter” and produces a message box containing it.

```
UCharT buf[2056];
IntT returnVal, index = 0;

returnVal =
  F_ApiCallClient("FrameSLT", "GetScriptParmByte---MyParameter");

while(returnVal > 100)
{
  buf[index++] = returnVal - 100;
  returnVal =
    F_ApiCallClient("FrameSLT", "GetScriptParmByte---MyParameter");
}

if(returnVal < 100)
  F_Sprintf(buf, "ERROR! The following error code was returned: %d",
    returnVal);
else buf[index] = 0;

F_ApiAlert((StringT)buf, FF_ALERT_CONTINUE_WARN);
```

Hello

Determines if FrameSLT is initialized and receiving external calls.

Syntax

```
F_ApiCallClient("FrameSLT", "Hello");
```

Usage description

`Hello` is a simple call to ensure that FrameSLT is available and responding to external calls.

Returns

`F_ApiCallClient()` returns one of the following values after a `Hello` call:

Value	Meaning
0	Communication with FrameSLT failed. Make sure that FrameSLT is initialized and running. Also, make sure that FrameSLT is properly registered in the `maker.ini` file under the name “FrameSLT.”
16	An evaluation copy of FrameSLT is installed, but the license is expired. External calls will not work.
17	Deprecated. This used to be the return for FrameSLT Lite, which no longer exists as of version 2.2.
18	FrameSLT installed and ready.

Syntax example

```
. . .
IntT returnVal;

. . .
returnVal = F_ApiCallClient("FrameSLT", "Hello");

if(returnVal < 17)
  F_ApiAlert("Error. FrameSLT is not ready.", FF_ALERT_CONTINUE_WARN);
```

ParseXPath

Parses an XPath expression, returning an internal sequence number for node queries.

Syntax

`F_ApiCallClient("FrameSLT", "ParseXPath---`*`Expression`*`---`*`ReportErrors`*`")`

where:

Expression	XPath expression to parse.
ReportErrors	Indicates whether to report parsing errors or not, either `True` or `False`. If you specify `True`, FrameSLT will produce the standard error report if a parsing error is encountered. Otherwise, the return value will indicate if a parsing error occurs, but you will not know the nature of the error.

Usage description

`ParseXPath` parses an XPath expression, and if successful, returns an internal sequence number that you will need for `FindNextNode` node queries. You can call `ParseXPath` for multiple expressions, and provided that you store the sequence numbers, you can perform independent queries based on any of them afterwards. In other words, subsequent `ParseXPath` calls start new internal sequences and do not delete any previously parsed data.

The returned sequence number is how FrameSLT identifies a parsed XPath expression and is a required argument for `FindNextNode` calls. Each unique expression that is parsed will return a unique sequence number. All parsed data will remain in memory unless cleared with an `AllocateNodeHandlers`. Normally, memory constraints should not be a concern, unless you are parsing an excessive amount of expressions such as a few hundred or more. In this case, you may consider periodic calls to `AllocateNodeHandlers` to free up some memory. Note, however, that an `AllocateNodeHandlers` call will delete all currently-parsed data and render any previously-retrieved sequence numbers invalid.

Note: All statements made thus far about parsed XPath data remaining in memory assume that you have your memory constraints set to a reasonable capacity. For more information on setting memory constraints, see *“Preferences”* on page 10.

Returns

`F_ApiCallClient()` returns one of the following values after a `ParseXPath` call:

Value	Meaning
0	Communication error with FrameSLT.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command
3	The XPath contains an error and could not be parsed. To find out the nature of the error, set *`ReportErrors`* to `True` to produce the error report.
Any other number over 100	A sequence number indicating that the parse was successful. This sequence number becomes an internal identifier for the parsed expression will be necessary to perform queries using `FindNextNode`.

Syntax example

```
sequenceNumber =
  F_ApiCallClient("FrameSLT", "ParseXPath---//Section/Body[1]---True");
```

ResetSequence

Resets a parsed, internal XPath sequence for reuse.

Syntax

```
F_ApiCallClient("FrameSLT", "ResetSequence---SequenceNumber");
```

where:

SequenceNumber	Valid sequence number for a previously-parsed XPath expression, as returned by a `ParseXPath` call.

Usage description

`ResetSequence` resets an internal XPath sequence such that it can be used for a new query. Once `ResetSequence` is called, you can begin a new query, using a different context node if desired. For more information on sequence behavior and context nodes, see *“FindNextNode”* on page 124.

As an example, consider the following XPath expression:

```
//Body
```

After this expression is parsed, the first `FindNextNode` call will find the first `Body` element in the document. The next call finds the next `Body` element, and so on. When you reset the sequence, however, `FindNextNode` begins again at the root, finding the first `Body` element again.

The concept of resetting a sequence is necessary because an XPath query works in a sequential, contextual manner, which has a definitive starting and ending point. Once `FindNextNode` has exhausted a sequence and reached the end, the only logical way to use the sequence again is to reset it entirely and resume the query at some specified starting context. If you run a subsequent `FindNextNode` pattern on the same XPath expression, same original structure, and same context, it will always find the same nodes as the previous run.

Returns

`F_ApiCallClient()` returns one of the following values after a `ResetSequence` call:

Value	Meaning
0	Reset was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string
2	Incorrect number of arguments sent with the command
4	Bad sequence number. Check to make sure you have sent the sequence number returned by `ParseXPath`.

RetrieveAttrMatch

Retrieves the index (+100) of an attribute as matched by an XPath expression, following a `FindNextNode` call.

Syntax

```
F_ApiCallClient("FrameSLT", "RetrieveAttrMatch---SequenceNumber");
```

Usage description

Note: The index returned by `RetrieveAttrMatch` is actually the index plus 100, in order to reserve the lower return numbers for error reporting. For any value returned by `RetrieveAttrMatch`, you should subtract 100 from it before using it as an index.

`RetrieveAttrMatch` retrieves the index of the attribute matched by the most recent call to `FindNextNode`. This call is provided for XPath expressions that match attribute nodes, because `FindNextNode` returns element IDs only. If the respective XPath expression does not match attribute nodes, this call will return 99, corresponding to an actual index of -1.

As an example, consider the following expression:

```
//Body/@AttributeA
```

This expression matches attribute nodes, not element nodes. Specifically, it matches attribute nodes named "AttributeA" on `Body` elements. For each match, though, `FindNextNode` will return the ID of the parent `Body` element only. Therefore, `RetrieveAttrMatch` allows you to retrieve the index of the matched attribute as well. It must be called before the next `FindNextNode`, because each `FindNextNode` call resets the value to the most recent match.

The index retrieved by `RetrieveAttrMatch` corresponds to the attribute index (+100) of the attribute as if it were stored in an F_AttributesT array, as if retrieved by F_ApiGetAttributes() on the element returned by `FindNextNode`.

Returns

`F_ApiCallClient()` returns one of the following values after a `RetrieveAttrMatch` call:

Value	Meaning
0	Communication error with FrameSLT
1	General syntax error in call string
2	Incorrect number of arguments sent with the command

Value	**Meaning**
4	Bad sequence number. Check to make sure you have sent the sequence number returned by `ParseXPath`.
Any number over 98	The attribute index, plus 100. For example, if the call returns 101, the actual index is 1. If the call returns 99, the actual index is -1, meaning that there actually is no index. 99 (1-) should only be returned if the XPath expression matches elements instead of attributes.

Code sample

The following code sample performs an XPath query and reports the values of the matched attribute(s):

```
. . .
IntT sequenceNum,
  index;
F_ObjHandleT elemId,
  docId;
F_AttributesT attrs;
UCharT fnnCall[64],
  ramCall[64];

. . .

/* Get the ID of the active document */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Parse the XPath expression, which will match all attributes */
/* of all Body elements */
sequenceNum =
  F_ApiCallClient("FrameSLT", "ParseXPath---//Body/@*---True");

/* Form the call strings we are going to send to FrameSLT to */
/* navigate with the XPath and retrieve the index of matched attributes */
F_Sprintf(fnnCall, "FindNextNode---%d---%d---0---0", sequenceNum, docId);
F_Sprintf(ramCall, "RetrieveAttrMatch---%d", sequenceNum);

/* Get the element ID of the first match of the xpath */
elemId = (F_ObjHandleT)F_ApiCallClient("FrameSLT", (StringT)fnnCall);

/* Run this loop for each match of the XPath */
while(elemId)
{
  /* Get the index of the matched attribute from the previous query */
  index = F_ApiCallClient("FrameSLT", (StringT)ramCall);

  /* Get the attributes from the parent element of the matched */
  /* attribute, within which we will be able to find the matched */
  /* attribute according to the index */
  attrs = F_ApiGetAttributes(docId, elemId);
```

```
        /* If the attribute has any values, report the first one */
        /* Remember that the returned index is the actual index plus 100 */
        if(index > 99 &&
           attrs.val[index - 100].values.len > 0)
             F_ApiAlert(attrs.val[index - 100].values.val[0],
               FF_ALERT_CONTINUE_WARN);
        else
          F_ApiAlert("<no value>", FF_ALERT_CONTINUE_WARN);

        /* Clear the F_AttributesT array */
        F_ApiDeallocateAttributes(&attrs);

        /* Get the next XPath match */
        elemId = (F_ObjHandleT)F_ApiCallClient("FrameSLT", (StringT)fnnCall);
      }
```

RetrieveFileMatch

Retrieves the object handle ID (F_ObjHandleT) of the file that contains the matched node, following a `FindNextNode` call. This call is only applicable for XPath queries that traverse different books and/or documents using special axes such as `fmbook::`. If a query never leaves the original file, this call will always return the ID of that file. Note that this file may be a book or a document, depending upon the nature of the query.

Syntax

```
F_ApiCallClient("FrameSLT", "RetrieveFileMatch---SequenceNumber");
```

Returns

`F_ApiCallClient()` returns one of the following values after a `RetrieveAttrMatch` call:

Value	Meaning
0	Communication error with FrameSLT
1	General syntax error in call string
2	Incorrect number of arguments sent with the command
4	Bad sequence number. Check to make sure you have sent the sequence number returned by `ParseXPath`.
Any number over 100	The file ID.

RunNWScript

Runs a Node Wizard script or script event.

Syntax

```
F_ApiCallClient("FrameSLT",
  "RunNWScript---ScriptName---EventNumber---File---DoReporting")
```

where:

`ScriptName`	Valid, case-sensitive name of a defined Node Wizard script. Spaces in script names are permitted.
`EventNumber`	Specific event to run, within the script. Within a script, individual events are numbered sequentially, starting at 1. If you want to run the whole script, specify zero (0).
`File`	File on which the script should run, as one of the following: • A document or book object handle in integer form (integer form of the FDK F_ObjHandleT type) • A fully-qualified path name of an open document or book In either case, the file must be currently open.
`DoReporting`	Indicates whether reporting activities should occur, either `True` or `False`. Reporting activities include message boxes that report script errors and run statistics. If you specify `False`, no message boxes or reports should appear at all, whether or not the script runs successfully.

Usage description

`RunNWScript` runs a Node Wizard script that you have already defined in the scripts settings file. The results should be identical to those as if the script were run with the scripts dialog at **FrameSLT > Node Wizard Scripts**. For more information on writing and managing Node Wizard scripts, see *"Node Wizard scripts"* on page 45.

Tip: FrameSLT will determine whether the file to be acted upon is a book or document, and adjust processing accordingly. If you are processing a book and one or more components are not currently open, they will be skipped by the script. If you have the *DoReporting* flag set to True, you will be prompted first, otherwise the script will simply proceed on any components that are open.

Returns

`F_ApiCallClient()` returns one of the following values after a `RunNWScript` call:

Value	Meaning
0	Script ran successfully. ***Note:*** The failure of individual events and element/attribute actions does not return a script failure. Therefore, a return of zero does not necessarily indicate that the script performed the actions you intended. It merely indicates that the script was found in the settings file and that no critical errors occurred during the script run process.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command
6	Bad file argument. An invalid document or book ID or filename was sent. Make sure the argument represents the ID or filename of a valid, open document or book.

Value	Meaning
8	Bad script name. Make sure the script name sent represents a script defined in the script settings file. Note that script names are case-sensitive.
9	The script failed to complete for an unknown reason. Possibilities include: • The script is marked as “inactive” in the scripts settings file. • The script run was cancelled in-progress by the user (perhaps you).

Syntax example

```
returnVal = F_ApiCallClient("FrameSLT",
  "RunNWScript---MyScript---0---67108880---True");
```

Code sample

The following sample runs a script named “MyScript” on all components of the active, book, if a book is active and all components are open:

```
. . .

F_ObjHandleT bookId;
IntT returnVal;
UCharT buf[64];

. . .

/* Get the ID of the active book, on which we will run the script */
bookId = F_ApiGetId(0, FV_SessionId, FP_ActiveBook);

if(bookId)
{
  /* Form the argument to sent to FrameSLT. We will be running */
  /* a script named "MyScript". */
  F_Sprintf(buf, "RunNWScript---MyScript---0---%d---False", bookId);

  /* Call FrameSLT to run the script */
  returnVal = F_ApiCallClient("FrameSLT", (StringT)buf);

  /* Report how things went */
  if(returnVal == 0)
    F_ApiAlert("Script ran OK.", FF_ALERT_CONTINUE_WARN);
  else
    F_ApiAlert("An error occurred.", FF_ALERT_CONTINUE_WARN);
}
```

SetAppParm

Sets a general application parameter.

Syntax

```
F_ApiCallClient("FrameSLT", "SetAppParm---Name---Value")
```

...where the following table describes valid parameter names and values:

Name	**Description**	**Options for *Value***
`EC_NWScriptsDoc`	Sets the currently-active Node Wizard Scripts document.	• A fully-qualified path, using forward or backslashes. If the file is not open, FrameSLT will open a hidden copy and allow access to the scripts within, much like the Node Wizard Scripts dialog box when the scripts document is not open • A filename. The document must be currently open. • An F_ObjHandleT document ID. The document must be open. Note that in all cases, this argument is not case-sensitive.
`EC_ReturnIDType`	Specifies the type of element ID returned by a `FindNextNode` call.	• **FDK** - An integer form of an FDK F_ObjHandleT handle. When calling FrameSLT from another API client, this form is normally the most convenient. • **UID** - An element unique ID (FP_Unique property). When calling FrameSLT from ExtendScript, it is normally much easier to convert this type of ID to an ES object by using the `GetUniqueObject()` document method.

Returns

`F_ApiCallClient()` returns one of the following values after a `SetAppParm` call:

Value	**Meaning**
0	Operation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command.
5	The specified Node Wizard Scripts document could not located, likely due to an invalid path or document ID. Check the syntax of the command.
20	Invalid parameter name.

Syntax example

```
returnVal =
  F_ApiCallClient("FrameSLT", "SetAppParm---EC_ReturnIDType---UID");
```

SetParam

Note: This command was originally implemented to handle parameter usage during transformation activities, which are scheduled for deprecation. If you want to set a parameter for a Node Wizard Script, see `SetScriptParm`.

Sets a parameter value for use during transformation or deletes all current parameter values.

Syntax

```
F_ApiCallClient("FrameSLT", "SetParam---Name---[Value]")
```

where:

Name	Parameter name, or `delete_all` to clear all currently-defined parameters. The preceding dollar sign ($) used when a parameter is referenced in a stylesheet is not required.
Value	(Optional) Parameter value. This must be a static string, not an XPath expression as supported by `FSLT_param` elements. If omitted, an empty string is assumed.

Usage description

`SetParam` allows you to define a parameter before performing a transformation. It has some similarity with comparable XSLT processes where a parameter is passed to a stylesheet before transformation, with the following important differences:

- A defined parameter is not specific to any stylesheet. Any parameters that are defined will apply to any subsequent transformation action with `TransformFile`.
- All parameter definitions remain in memory until cleared with this command or a manual transformation is run (through the FrameSLT menu). These parameter definitions are not used for manual transformations and will be cleared out by a manual action.
- Setting a parameter to an empty string does not delete its definition; rather, it simply defines it as an empty string.
- Similar (in some respects) to XSLT, this command will override any `FSLT_param` elements in the stylesheet(s) that define the same parameter. In other words, an `FSLT_param` element will be ignored during an external-call transformation if previously-defined with this command. Unlike XSLT, however, a stylesheet does not need to contain a matching `FSLT_param` element at all if the parameter is defined by this command and transformed with `TransformFile`.

For more information on parameter usage in stylesheets, see:

- *“About parameters in XPath expressions”* on page 85
- *“Use of parameters in source file paths”* on page 81
- `FSLT_value-of`

Returns

`F_ApiCallClient()` returns one of the following values after a `SetParam` call:

Value	Meaning
0	Operation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn’t work, consider calling `Hello` to verify that FrameSLT is active.

Value	Meaning
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command

Syntax examples

```
returnVal =
  F_ApiCallClient("FrameSLT", "SetParam---MyParameter---SomeValue");
returnVal =
  F_ApiCallClient("FrameSLT", "SetParam---delete_all");
```

SetScriptParm

Sets a parameter value for use during a Node Wizard script.

Note: If you are using ExtendScript, you can also set a parameter using an external object. For more information, see *"Using FrameSLT as an ExtendScript external object"* on page 149.

Syntax

```
F_ApiCallClient("FrameSLT", "SetScriptParm---Name---Value")
```

where:

Name	Parameter name. The preceding dollar sign ($) sometimes used for parameter notation is not required.
Value	Value to set. This may be any string, including an empty string.

Usage description

`SetScriptParm` sets a parameter for use by Node Wizards Scripts, much like a `Set_parameter` action within a script. Its usage is otherwise intuitive, noting that:

- When you set a parameter with this call, it persists in memory until a script changes it or you call `ClearScriptParms`. Unlike a parameter set by a script, a parameter set with this call does not get cleared when a new script is launched.
- Once set, a parameter is available for any script.

Returns

`F_ApiCallClient()` returns one of the following values after a `SetScriptParm` call:

Value	Meaning
0	Operation was successful. ***Note:*** 0 is also returned if a communication error occurs with FrameSLT. If you suspect that the command didn't work, consider calling `Hello` to verify that FrameSLT is active.
1	General syntax error in call string.
2	Incorrect number of arguments sent with the command

Syntax example

```
returnVal =
  F_ApiCallClient("FrameSLT", "SetScriptParm---MyParameter---SomeValue");
```

TransformFile

Transforms a book or document, based on a file name or object ID sent with the command.

Syntax

```
    F_ApiCallClient("FrameSLT",
        "TransformFile---StylesheetFile---DupeDocPath---ReportErrors");
```

where:

StylesheetFile	Stylesheet file or book of files to be transformed, as one of the following: • A document or book object handle in integer form (integer form of the FDK F_ObjHandleT type) • A fully-qualified path name of an open document or book In either case, the file must be currently open.
DupeDocPath	The fully-qualified path for the duplicated, transformed file, applicable only for transforming books. Only books are directed to a new folder during a “duplicate file” transform. To indicate a duplicate file transform on a single document, specify `DUPE`. A duplicate file will be created, but not saved to a new folder. For books and documents, to indicate a “source file” transform, specify `NULL`. Note that a source file transform operates directly on your source files and should be performed with caution.
ReportErrors	Indicates whether to report errors or not, either `True` or `False`. If you specify `True`, FrameSLT will produce the standard error report if errors are encountered. If you specify `False`, the return value will indicate if an error occurs, but you may not know the nature of the error.

Usage description

`TransformFile` performs a full transformation of the specified book or document. The stylesheet file(s) for transformation must be currently open, because this command will not open any stylesheets. It may open source files to retrieve content, but only if your FrameSLT preferences are set up to allow this. For more information, see *“Preferences”* on page 10.

`TransformFile` allows you to specify a target file path, if you wish to duplicate the file prior to transformation. The actual syntax of the path is only relevant for book transformations, because duplicate document transformations always create a duplicate in the same folder and apply the filename addendum specified in your preferences. If you want to duplicate a single document, simply specify any string other than `NULL`. For either a document a book, specification of `NULL` will cause the transformation to occur on the source document.

If the transformation is successful, this call returns an integer form of the transformed document or book ID. In the case of a duplicate document transformation, this will be the ID of the new, duplicated document and will be different than the ID you sent in the original call. For book transformations, the ID should be the same, but keep in mind that if you duplicated the book, it is not the same book you started with. It is your responsibility to handle that document or book afterwards. This command does not open, save, or close any files, except for source files opened by the transformation itself, as applicable.

Tip: Before transformation, you can define parameter values with `SetParam`, applicable if your stylesheets use parameters.

Returns

`F_ApiCallClient()` returns one of the following values after a `TransformFile` call:

Value	Meaning
`0`	Communication error with FrameSLT
`1`	General syntax error in call string
`2`	Incorrect number of arguments sent with the command
`6`	Bad stylesheet file argument. An invalid document or book ID or filename was sent. Make sure the argument represents the ID or filename of a valid, open document or book.
`11`	Could not duplicate the document. You have attempted to transform a document by making a duplicate first, but the duplication process failed. This may occur for any number of reasons, so you may consider working with the file manually to see if it has any overt problems. If the original stylesheet document has unsaved changes, FrameSLT attempted to save it before duplication, so the problem may have occurred at that point due to server, network, or permission errors.
`12`	Could not find structure in the stylesheet document. FrameSLT was unable to find any structure in the main flow of the document, so no transformation actually took place.
`13`	Pre-processing of the stylesheet document failed. Before transformation, FrameSLT performs a variety of preprocessing activities on the stylesheet, such as parsing XPath expressions and validating transformation element setups. Any single failure can cause the process to abort. You can learn the specific nature of these errors by setting ***`ReportErrors`*** to `True`, allowing FrameSLT to generate its error report.
`14`	General transformation error. FrameSLT encountered an unrecoverable error during the transformation of a stylesheet document, and aborted the transformation. You can learn the specific nature of these errors by setting ***`ReportErrors`*** to `True`, allowing FrameSLT to generate its error report.
`15`	Error during book preparation. FrameSLT performs a set of preliminary steps to prepare a book before the actual transformation begins. An error during this process is usually unrecoverable and causes the transformation to abort. You can learn the specific nature of these errors by setting ***`ReportErrors`*** to `True`, allowing FrameSLT to generate its error report. ***Note:*** If you are attempting to transform a book, and you have provided a folder path for duplication of the book, a single mistake in the path will cause this error. Make absolutely sure that the path for the new book is exactly correct. For more information on specifying this path, see *"`TransformFile` syntax examples"* on page 142.
Any number greater than 15	An integer form of the transformed document or book ID. If you transformed a source file, this ID should be the same as the ID you sent with `TransformFile`. A returned ID generally indicates that the transformation was successful.

`TransformFile` syntax examples

`TransformFile` calls are syntactically challenging because of the potential presence of file paths, which contain backslashes. In a string in C, backslashes must be sent as an escape sequence, represented by a double backslash (`\\`). For example, the following are some examples of `TransformFile` calls:

```
//Document transform by ID, source file
F_ApiCallClient("FrameSLT",
  "TransformFile---1842312---NULL---True");
//Document transform by ID, duplicate doc
F_ApiCallClient("FrameSLT",
  "TransformFile---1842312--Dupe---True");
//Document transform by file name, source file
F_ApiCallClient("FrameSLT",
  "TransformFile---C:\\MyDocs\\Stylesheet.fm---NULL---True");
//Book transform by ID, source file
F_ApiCallClient("FrameSLT",
  "TransformFile---3425343---NULL---True");
//Book transform by ID, duplicate book
F_ApiCallClient("FrameSLT",
  "TransformFile---3425343---C:\\MyXformedDocs\\---True");
//Book transform, duplicate book
F_ApiCallClient("FrameSLT",
  "TransformFile---C:\\MyDocs\\MyBook.book---C:\\MyXformedDocs\\---True");
```

`TransformFile` code samples

The following example shows the basic syntax of an actual `TransformFile` call, in C FDK format, to transform the active document without duplicating it.

```
. . .
F_ObjHandleT docId;
UCharT arg[100];
IntT returnVal;

. . .
/* Get a document ID */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Form the argument for the FindNextNode call */
F_Sprintf(arg, "TransformFile---%d---NULL---True", docId);

/* Call FrameSLT to transform the file */
returnVal = F_ApiCallClient("FrameSLT", (StringT)arg);

/* Report */
if(returnVal > 15)
  F_ApiAlert("Transformation successful.", FF_ALERT_CONTINUE_WARN);
else
  F_ApiAlert("Transformation failed. Please see the report.",
    FF_ALERT_CONTINUE_WARN);

. . .
```

The following code represents the same functionality in FrameScript format:

Note: Many thanks to Rick Quatro of Carmen Publishing, www.frameexpert.com, for this FrameScript translation of the previous FDK code sample.

```
// Get the ID of the active document.
Set docId = ActiveDoc;
If(docId = 0)
  MsgBox 'There is no active document.' Mode(Warn);
  LeaveSub;
EndIf

// Convert the document ID to an integer.
New Integer NewVar(docInt) Value(docId);

// Form the argument for the TransformFile call.
Set arg = 'TransformFile---'+docInt+'---NULL---True';

// Call FrameSLT to transform the active document.
CallClient FrameClient('FrameSLT') Message(arg) ReturnVal(iReturnVal);

// Report
If iReturnVal > 15
  MsgBox 'Transformation sucessful.' Mode(Note);
Else
  MsgBox 'Transformation failed. Please see the report.'
    Mode(Note);
EndIf
```

Detailed example—Calling FrameSLT (FDK)

The following example contains C language code for use with the FDK. For the same sample in FrameScript form, see *"Detailed example—Calling FrameSLT (FrameScript)"* on page 146.

This example uses two different XPath expressions to find Emphasis elements and apply strikethrough text to them. The two XPath expressions used are:

```
//Section/Body
Emphasis
```

In summary, the first expression is used to find Body elements that are children of Sections. Then, it uses the second expression to find Emphasis element children of those Body elements. Note that these two XPath expressions could be combined into one; however, this example uses them separately to demonstrate most available FrameSLT calls, and to show how two different XPath expressions can be used simultaneously.

If you want to use this sample function on a different structure, you can simply change the XPath expressions as appropriate.

Tip: FrameSLT includes a sample file, `External_Calls_Sample.fm`, designed to work with this code. If installed correctly, you should find it in your `SampleFiles` folder.

```
VoidT FrameSLT_Sample_Calls()
{
   F_ObjHandleT docId,
    bodyElemId,
    emphasisElemId;

  UIntT xPathSequence1,
    xPathSequence2,
    returnVal;

  UCharT sequence1Arg[64],
    sequence2Arg[64],
    resetSequenceArg[64];

  F_TextRangeT tr;

  F_PropValT strikethroughProp;

  /* Get the ID of the active document */
  docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
  if(!docId)
  {
    F_ApiAlert("No active document", FF_ALERT_CONTINUE_WARN);
    return;
  }

  /* Parse the XPath expressions and retrieve the sequence numbers */
  xPathSequence1 = F_ApiCallClient("FrameSLT",
    "ParseXPath---//Section/Body---True");
  xPathSequence2 = F_ApiCallClient("FrameSLT",
    "ParseXPath---Emphasis---True");

  /* If a parsing error occurred, the return value will be
   * less than 21 (20 is the highest error code) */
  if(xPathSequence1 < 21 || xPathSequence2 < 21)
  {
    F_ApiAlert("Error parsing XPath", FF_ALERT_CONTINUE_WARN);
    return;
  }

  /* Set up the text property structure for applying strikethrough text.
   * Used later. */
  strikethroughProp.propIdent.num = FP_Strikethrough;
  strikethroughProp.propVal.valType = FT_Integer;
  strikethroughProp.propVal.u.ival = True;

  /* Set up the argument for the first FindNextNode call, for the
   * first XPath expression. No context node ID is necessary because
   * the XPath begins with a "go-to-root" axis. */
  F_Sprintf(sequence1Arg, "FindNextNode---%d---%d---0",
    xPathSequence1, docId);

  /* Set up the argument that will be used later to reset the second
   * XPath sequence */
  F_Sprintf(resetSequenceArg, "ResetSequence---%d", xPathSequence2);

  /* Make an initial call to find the first applicable Body element */
  bodyElemId = F_ApiCallClient("FrameSLT", (StringT)sequence1Arg);
```

```
    /* Launch the "outer loop", which will step through the main
     * flow looking for Body elements that are children of Sections. */
    while(bodyElemId > 20)
    {
      /* Reset the second XPath sequence, in preparation for a new
       * query, using the current Body element as the context node */
      returnVal = F_ApiCallClient("FrameSLT", (StringT)resetSequenceArg);
      if(returnVal != 0)
      {
        F_ApiAlert("Error resetting sequence", FF_ALERT_CONTINUE_WARN);
        break;
      }

      /* Set up the argument to query with the second sequence, using the
       * current Body element as the context */
      F_Sprintf(sequence2Arg, "FindNextNode---%d---%d---%d",
        xPathSequence2, docId, bodyElemId);

      /* Make an initial call to find the first applicable Emphasis element */
      emphasisElemId = F_ApiCallClient("FrameSLT", (StringT)sequence2Arg);

      /* Step through all emphasis children, applying strikethrough text */
      while(emphasisElemId > 20)
      {
        /* Get the text range of the Emphasis element */
        tr = F_ApiGetTextRange(docId, emphasisElemId, FP_TextRange);
        /* Apply strikethrough text */
        F_ApiSetTextPropVal(docId, &tr, &strikethroughProp);

        /* Query for the next Emphasis element */
        emphasisElemId = F_ApiCallClient("FrameSLT", (StringT)sequence2Arg);
      }

      /* Back in the main loop, query for the next Body element*/
      bodyElemId = F_ApiCallClient("FrameSLT", (StringT)sequence1Arg);

    } /* End main loop */

    /* Free the strikethrough PropVal structure */
    F_ApiDeallocatePropVal(&strikethroughProp);
  }
```

Detailed example—Calling FrameSLT (FrameScript)

Note: Many thanks to Rick Quatro of Carmen Publishing, www.frameexpert.com, for this FrameScript translation of the previous FDK code sample.

The following example contains FrameScript code. For the same sample in C language form, see *"Detailed example—Calling FrameSLT (FDK)"* on page 144.

This example uses two different XPath expressions to find Emphasis elements and apply strikethrough text to them. The two XPath expressions used are:

```
//Section/Body
Emphasis
```

In summary, the first expression is used to find Body elements that are children of Sections. Then, it uses the second expression to find Emphasis elements within those Body elements. Note that these two XPath expressions could be combined into one; however, this example uses them separately to demonstrate all available FrameSLT calls, and to show how two different XPath expressions can be used simultaneously.

If you want to use this sample function on a different structure, you can simply change the XPath expressions as appropriate.

Tip: FrameSLT includes a sample file, `External_Calls_Sample.fm`, designed to work with this script. If installed correctly, you should find it in your `SampleFiles` folder.

```
// Get the ID of the active document.
Set docId = ActiveDoc;
If(docId = 0)
  MsgBox 'There is no active document.' Mode(Warn);
  LeaveSub;
EndIf

// Convert the document ID to an integer.
New Integer NewVar(docInt) Value(docId);

// Parse the XPath expressions and retrieve the sequence numbers.
CallClient FrameClient('FrameSLT')
  Message('ParseXPath---//Section/Body---True')
  ReturnVal(xPathSequence1);
CallClient FrameClient('FrameSLT')
  Message('ParseXPath---Emphasis---True')
  ReturnVal(xPathSequence2);
// If a parsing error occurred, the return value will be less
// than 21 (20 is the highest error code).
If (xPathSequence1 < 21) or (xPathSequence2 < 21)
  MsgBox 'Error parsing XPath.' Mode(Warn);
  LeaveSub;
EndIf

// Make a property list to be used for applying the strikethrough text.
New PropertyList NewVar(strikethroughProp)
  Strikethrough(True);

// Set up the argument for the first FindNextNode call, for the
// first XPath expression. No context node ID is necessary
// because the XPath begins with a "go-to-root" axis.
Set sequence1Arg = 'FindNextNode---'+xPathSequence1+'---'+
  docInt+'---0';

// Set up the argument that will be used later to reset the second
// XPath sequence.
Set resetSequenceArg = 'ResetSequence---'+xPathSequence2;

// Make an initial call to find the first applicable Body element.
CallClient FrameClient('FrameSLT') Message(sequence1Arg)
  ReturnVal(bodyElemId);

// Launch the "outer loop" which will step through the main flow
// looking for  Body elements that are children of Sections.
Loop While(bodyElemId > 20)

  // Reset the second XPath sequence, in preparation for a new query,
  // using the current Body element as the context node.
  CallClient FrameClient('FrameSLT') Message(resetSequenceArg)
    ReturnVal(returnVal);

  If returnVal not= 0
    MsgBox 'Error resetting sequence.' Mode(Warn);
    LeaveLoop;
  EndIf

  // Set up the argument to query with the second sequence, using the
  // current Body element as the context.
  Set sequence2Arg = 'FindNextNode---'+xPathSequence2+'---'+
    docInt+'---'+bodyElemId;

  // Make an initial call to find the first applicable Emphasis element.
  CallClient FrameClient('FrameSLT') Message(sequence2Arg)
    ReturnVal(emphasisElemId);

  // Step through all Emphasis children, applying strikethrough text.
  Loop While(emphasisElemId > 20)
    New Object NewVar(emphasisElemId) IntValue(emphasisElemId)
      DocObject(docId);
    // Apply the strikethrough properties.
    Apply TextProperties TextRange(emphasisElemId.TextRange)
      Properties(strikethroughProp);

    // Find the next Emphasis element.
    CallClient FrameClient('FrameSLT') Message(sequence2Arg)
      ReturnVal(emphasisElemId);

  EndLoop

  // Back in the main loop, query for the next Body element.
  CallClient FrameClient('FrameSLT') Message(sequence1Arg)
    ReturnVal(bodyElemId);
EndLoop
```

Using FrameSLT as an ExtendScript external object

This release implements some preliminary functionality related to ExtendScript (FrameMaker 10 and later) and the use of FrameSLT as an “external object” (EO). This functionality is currently limited to setting and retrieving Node Wizard Script parameter values.

In brief, the EO architecture allows you to invoke methods (functions) directly from the FrameSLT code and receive a simple data type (string, double, integer, or boolean) as a return. The use of EOs versus `CallClient()` has a variety of advantages, including:

- A simpler syntax for calling the plugin
- The ability to retrieve strings, versus CallClient() which supports an integer return only

As a side note, the limited current FrameSLT EO support was inspired by the difficulty of returning a string parameter value with `CallClient()`. Having seen it work, we believe that this will eventually expand to cover all functionality that FrameSLT has exposed to `CallClient()`, and perhaps more. If you have any comments or suggestions on this subject, we would love to hear them, while we evaluate the possibilities with EOs and the direction in which we want to take them.

Current EO method support includes two simple functions related to Node Wizard Script parameters:

`setScriptParm(`*`parmName`*`, `*`parmVal`*`)`

`getScriptParm(`*`parmName`*`)`

...where all arguments are strings. As an example, the following script produces an alert box with the text “Hello world”:

```
var framesltDLLPath = "C:\\Program Files\\Adobe\\AdobeFrameMaker10" +
                       "\\WestStreet\\FrameSLT_FM10.dll";

var framesltEO = new ExternalObject("lib:" + framesltDLLPath);

framesltEO.setScriptParm("MyParm", "Hello world!");
var val = framesltEO.getScriptParm("MyParm");
alert(val);
```

Note the following:

- The instantiation of the EO requires the correct path to the `FrameSLT.dll` file.
- When setting and retrieving parameter values with an EO, the notes under `SetScriptParm` and `GetScriptParmByte` about general parameter behavior are still applicable.
- If `getScriptParm()` is called for an undefined parameter, an error string is returned. This string is defined by a setting in your local preferences and is set to “`PARAMETER_NOT_DEFINED`” from the factory.