ptc

arbortext®

Customizer's Guide

8.3.3.0

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

Contents

About This Guide

The *Customizer's Guide* provides detailed instructions on how to configure and customize Arbortext Editor features for use at your site. Examples of typical customizations are provided throughout the guide to illustrate steps you'll take to configure Arbortext Editor to address your specific needs. The *Customizer's Guide* is a companion to the *Programmer's Reference*, available in the Arbortext Editor Help Center.

The information covered in the *Customizer's Guide* is divided as follows:

- *About This Guide* — An introduction to this guide and the information it covers.
- *Custom applications* — An overview of implementing custom applications with Arbortext products.
- *Customizing your site's profiling configuration* — Instructions on configuring and customizing profiling to be specific for your site.
- *Customizing help* — Details on how to update Arbortext online help to be specific to your site.
- *Customizing PDF publishing* — Information on configuring and customizing your site's PDF publishing capabilities.
- *Customizing Publishing Rules* — Information on customizing publishing rules, rule sets, and rule files.
- *Working with XUI (XML User Interface) dialog boxes* — Instructions on creating, displaying, and manipulating dialog boxes in real time by writing and modifying XML documents.

- *Working with ActiveX controls* — Instructions on defining and implementing ActiveX controls at your site.
- *Merging data from other sources* — An overview of the Arbortext data merging capabilities and references to other sources of information.
- *Working with Arbortext Import/Export* — An overview of using Arbortext Import/Export, configuration instructions, a description of the Arbortext Import/Export API, instructions on using Arbortext Import/Export in batch mode, migration information, and troubleshooting information.
- *Customizing copying and pasting from other applications* — Details on how to customize the use of Arbortext Import/Export to paste content from other applications as tags conforming to a document type.
- *Customizing DITA support* — Details on how to customize the Arbortext Editor user interface for editing DITA documents.

Prerequisite Knowledge

The *Customizer's Guide* assumes advanced skill using Java, JavaScript, JScript, VBScript, or COM (Component Object Model). If you're creating an Arbortext Publishing Engine application, you also need to be familiar with Java servlets, servlet containers, web servers, the HTTP protocol, and the SOAP protocol.

Arbortext Editor and Arbortext Publishing Engine supporting documentation and related Javadoc can be found in the Arbortext Editor Help Center. Arbortext Command Language (ACL) documentation is included in the Help Center, and is not the focus of the *Customizer's Guide*.

If you are looking for more general information on programming or scripting languages, you may want to consult the following resources:

- *Thinking in Java*, by Bruce Eckel. Published by Prentice Hall PTR.
- Oracle has extensive Java information available at its web site www.oracle.com/technetwork/java/index.html. The tutorials are especially helpful to beginners.
- *JavaScript: The Definitive Guide*, by David Flanagan. Published by O'Reilly and Associates Inc.
- Mozilla has extensive JavaScript information available at its web site www.mozilla.org.
- ECMA International (European Computer Manufacturers Association) has the *ECMAScript Language Specification*, which is the standard used for JavaScript, available at its web site www.ecma.ch.
- Microsoft has extensive information about JScript, VBScript, ActiveX scripting host, and COM available at its web site msdn.microsoft.com.

1

Custom Applications

Overview of Custom Programs and Scripts

The Arbortext Editor and Arbortext Publishing Engine installations have directory structures within them where you can place your custom scripts and programs. The `custom` and the `application` directories are described in the following sections.

The Custom Directory Structure

The *`Arbortext-path`*`\custom` directory has a subdirectory structure designed to hold your custom programs and scripts and make them automatically available during the session. At startup, these subdirectories are searched for Java, JavaScript, JScript, VBScript, ACL, and composer configuration files. You can also provide custom document types, entities, fonts, graphics, and native shared libraries and DLLs. The supported file types are automatically accessed if they reside in the appropriate subdirectory. Implementing your custom files using this approach takes advantage of the startup sequence to automatically locate your custom files. The *`Arbortext-path`*`\custom` directory and its subdirectories are explained in detail in this chapter.

The Application Directory Structure

The *`Arbortext-path`*`\application` subdirectory can contain custom applications as well as application software distributed by Arbortext. The `application` directory must have one or more uniquely named subdirectories, each containing a specific configuration file, `application.xml`, that conforms to a specific format. At startup, the `application` directory is searched for subdirectories and the presence of a valid `application.xml` file. In the uniquely named subdirectory, all subdirectories of the `custom` directory are supported. The custom application in a `application` then uses these subdirectories in the same way as the `custom` directory structure. You can also have additional subdirectories needed to support the implementation of this type of custom application. Implementing your custom application using this approach takes advantage of the startup sequence, supports delivering a completely self-contained custom application, and offers the option of setting the conditions for whether the application should be loaded. The `application` directory is also explained in this chapter.

Description of the Custom Directory Structure

When Arbortext Editor or an Arbortext PE sub-process starts, it can access custom files placed in specific directories. At startup, it automatically looks for compiled Java files (`.class` and `.jar` files), JavaScript, JScript, VBScript, ACL, document type, publishing configuration and other types of files within the *`Arbortext-path`*`\custom` directory structure.

You can have one or more `custom` directories outside the *`Arbortext-path`* install tree. To specify a path list for their locations, set the *APTCUSTOM* environment variable. The `custom` directory must be located using a file system; HTTP references are not supported.

At startup, some search paths are automatically prepended with the path to a `custom` subdirectory. Startup automatically sets some of these search paths using a symbolic variable as a path specification. You can use symbolic parameters to represent a search path in the context of the default search path, the location of the install tree, or the locale.

If a directory supports more than one type of file, the file types are processed in the following order:

- `.acl` (Arbortext Command Language) files
- `.js` (JavaScript or JScript) files
- `.class` (Java) files
- `.vbs` (VBScript) files

For each file type, its files are processed in alphabetical order by file name.

The *`Arbortext-path`*`\custom` directory is processed at startup. If you add custom applications and document types after startup, they're not recognized during the session. If you're using Arbortext Editor, it needs to be closed and restarted. If you're using Arbortext Publishing Engine, you need to stop and restart the Arbortext Publishing Engine to re-initialize the Arbortext PE sub-processes.

custom.xml File

At the top level of the `custom` directory is the `custom.xml` file. Following is the default version of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Arbortext, Inc., 1988-2009, v.4002-->
<ApplicationConfiguration
 xmlns="http://www.arbortext.com/namespace/doctypes/appcfg">
 <Information>
 <!--The following name will be shown in the New dialog
 as the category for all document types in this
 custom directory that do not specify a category.-->
```

```
 <Name>Custom Directory Name</Name>
 </Information>
</ApplicationConfiguration>
```

This file is only used when you have a custom document type in the `custom\doctypes` subdirectory, and you have not designated a category name for the document type in the associated document type configuration (`.dcf`) file's `NewDialog` element. In this case, the name in the `custom.xml` file's `Name` element is used as the **Category** name for the document type(s) in the `custom\doctypes` subdirectory in the **New Document** dialog box.

Subdirectory Structure

The following list describes each `custom` subdirectory and how it's used. Arbortext Editor and Arbortext Publishing Engine look in these directories for any references that use a relative path or have no specified path.

- `classes` subdirectory

  Holds compiled Java `.class` and `.jar` files.

  TheArbortext Editor and Arbortext Publishing Engine JVM Java class path holds a list of directories and paths to `.jar` files. Any files matching `*.jar` are prepended to the JVM Java class path. Then the `classes` parent directory is prepended, putting it first in the JVM Java class path.

  In cases where a class file occurs in more than one `.jar` file, you can extract the preferred `.class` file from its `.jar` file and place it in a subdirectory path of the `classes` directory to control which one takes precedent.

- `composer` subdirectory

  Holds publishing configuration files (`.ccf`, `.ent`, and `.xml` files) and can support a `catalog` file. Supports one level of subdirectories.

  The default path is *`Arbortext-path`*`\composer`. If there are any subdirectories of the `custom\composer` directory, those subdirectories are prepended to the publishing configuration path. Then the `custom\composer` parent directory is prepended to the path. If the `custom\composer` directory contains a `catalog` file, that directory is also prepended to the catalog path.

- `dialogs` subdirectory

  Holds dialog files that can be accessed from custom applications, such as one that uses the AOM `Application.createDialogFromFile` method.

  The *`Arbortext-path`*`\samples\XUI\preferences\pref_exts.zip` contains a sample application that adds a tab to the Preferences window as a way to extend preferences for custom applications. Refer to the `readme.txt` file for more information.

If there are any subdirectories of the `custom\dialogs` directory, those subdirectories are prepended to the dialog path. Then the `custom\dialogs` parent directory is prepended to the dialog path.

- `ditarefs` subdirectory

  Holds content referenced by DITA documents when the reference is not specified as either an absolute path name or a path name relative to the current document directory. For example, the `ditarefs` subdirectory could hold content referenced by topic references, content references, and so forth. Supports one level of subdirectories.

  The default DITA reference path is *`Arbortext-path`*`\ditarefs`. The DITA references path can be set in the **File Locations** category of the **Tools ▸ Preferences** dialog box. You can also use the `set ditapath` option or the *APTDITAPATH* environment variable to set the default path for DITA references. If there are any subdirectories of the `custom\ditarefs` directory, those subdirectories are prepended to the path. Then the `custom\ditarefs` parent directory is prepended to the path.

  Note

  Graphic references from DITA documents are resolved using the graphics path list.

- `dictionaries` subdirectory

  Holds user-defined dictionary files that can be used by the spelling checker. Supports one level of subdirectories.

  The default path is *`Arbortext-path`*`\lib\proximity\userdict`. If there are any subdirectories of the `custom\dictionaries` directory, those subdirectories are prepended to the dictionary path. Then the `custom\dictionaries` parent directory is prepended to the dictionary path.

- `doctypes` subdirectory

  Holds a custom `catalog` file and document type files. Supports one level of subdirectories. Each document type should reside in a uniquely named subdirectory of `doctypes`. The subdirectory should also contain a `catalog` file for the custom document type. A `doctypes` subdirectory can also contain a subset of the complete document type file set. You can place a document type configuration file `.dcf` or stylesheets in a `\custom\doctypes\`*`doctype`* directory.

  You can add a stylesheet to the list of stylesheets that displays when you make a publishing request using one of the **File ▸ Publish** choices. Arbortext Editor

and Arbortext Publishing Engine search each `\custom\doctypes\`*`doctype`* directory and aggregate the list of stylesheets. For example, you can add stylesheets for the asdocbook built-in document type (`asdocbook`) by placing them in *`Arbortext-path`*`\custom\doctypes\asdocbook`.

If a document does not specify an Editor view stylesheet with a stylesheet association PI, Arbortext Editor will first search first the document directory, then the relevant `\custom\doctypes\`*`doctype`* directory, and finally the original location for the *`doctype`* directory.

If the subdirectory contains only a `.dcf` file, it must conform to a naming convention that expects the subdirectory and `.dcf` file name to reflect the base document type name. For example, you could customize the default asdocbook `asdocbook.dcf` file, and put it in *`Arbortext-path`*`\custom\doctypes\asdocbook\asdocbook.dcf` to override the built-in `.dcf`. Note that the document type subdirectory and file name must be the same as the default document type name for Arbortext Editor and Arbortext Publishing Engine to find all the relevant document type files.

A DCF file can reference other files, such as the `.pcf`, `demo.xml`, and `template.xml` files. Custom versions of these files can be placed with the `.dcf` in `\custom\doctypes\`*`doctype`*. If Arbortext Editor and Arbortext Publishing Engine find a `.dcf` in the `\custom\doctypes\`*`doctype`* location, relative path references are resolved by first searching the same directory as the `.dcf` and then by searching the document type directory in the original location.

The default catalog path is *`Arbortext-path`*`\doctypes`. If there are any subdirectories of the `custom\doctypes` directory that contain a `catalog` file, those subdirectories are prepended to the catalog path. Then the `custom\doctypes` parent directory is prepended to the catalog path.

You can place custom tag template files (`.tpl`) in a `custom\doctypes\`*`doctype`*`\tagtemplates` directory. The `custom\tagtemplates` directory can also be used as a more generally available location for tag templates.

Any document type from the `custom\doctypes` directory is also added to the list of available document types that are displayed in the **File ▸ New** dialog box.

- `entities` subdirectory

  Holds file entities. Supports one level of subdirectories.

  A file entity is any structurally complete document unit saved as a file. File entities commonly have an `.xml` file extension.

The default entity path is *Arbortext-path*\entities. If there are any subdirectories of the custom\entities directory, those subdirectories are prepended to the entity path. Then the custom\entities parent directory is prepended to the entities path.

- fonts subdirectory

  Holds custom AFM or TFM font metric files (.afm and .tfm).

  The default fonts path is *Arbortext-path*\fonts. If there are fonts in custom\fonts, the path is prepended. If the *APTTEXFONTS* environment variable is set, the custom\fonts directory is prepended to it.

- formats subdirectory

  Holds custom PubTex format files (.fmt).

  The default PubTex format path is *Arbortext-path*\formats. If there are .fmt files in custom\formats, the path is prepended. If the *APTTEXFMTS* environment variable is set, the custom\formats directory is prepended to it.

- framesets subdirectory

  Holds custom framesets for **Publish ▸ For Web**. Supports one level of subdirectories. Framesets are defined in the document type configuration file.

  The default frameset path is *Arbortext-path*\framesets. If there are any subdirectories of the custom\framesets directory, those subdirectories are prepended to the framesets path. Then the custom\framesets parent directory is prepended to the frameset path.

- graphics subdirectory

  Holds graphic files. Supports one level of subdirectories.

  The default graphics path is *Arbortext-path*\graphics. If there are any subdirectories of the custom\graphics directory, those subdirectories are prepended to the graphics path. Then the custom\graphics parent directory is prepended to the graphics path.

- importexport subdirectory

  Holds Arbortext Import/Export Import project files.

- inputs subdirectory

  Holds source files for custom macros, program fixes, or other customizations in a custom.tmx. Refer to Using .tmx files for more information. Document type and document .tmx files can be placed in the custom\doctypes directory.

Also holds `.tex` files and source files for hyphenation exception and pattern rules in `.exc` and `.pat` files.

The default source path is *`Arbortext-path`*`\inputs`. Then the *`Arbortext-path`*`\custom\inputs` directory is prepended to it.

- `lib` subdirectory

  Holds custom versions of the `.pdfcf` PDF configuration file. The default path for `.pdfcf` files is *`Arbortext-path`*`\lib`. Then the *`Arbortext-path`*`\custom\lib` directory is prepended to it. For more information on creating `.pdfcf` files, refer to the *Customizer's Guide*.

  In addition, the `lib` subdirectory can hold `.wcf` files for custom window classes. For more information on creating `.wcf` files for window classes, refer to the *Creating custom window class preferences files* in the Arbortext Editor help.

  The `lib` subdirectory can also hold custom versions of the following files:

  `charent.cf`

  `charmap.cf`

  `installprefs.acl`

  `prted.pro`

  `pubview.cf`

  `pubview.fnt`

  `tfmfont.cf`

  `tfmscaling.cf`

  `tfontsub.cf`

  `wcharset.cf`

  `wfontsub.cf`

  `xcharset.cf`

  `xfontsub.cf`

  You can specify more than one `charent.cf` file, as the effects are cumulative. Refer to the *Setting paths for new character set files* and *APTCUSTOM environment variable* topics in the online help for more information.

  The `custom\lib` directory also has `locale\`*`locale-name`* subdirectories. The default path is the appropriate locale subdirectory of

Arbortext-path\lib\locale. The locale-specific subdirectory of the custom\lib\locale directory is prepended to the default locale path.

The locale*locale-name* can hold custom versions of the .pdfcf PDF configuration file. For more information on creating .pdfcf files, refer to the *Customizer's Guide*.

Each locale*locale-name* directory can hold custom versions of the following files:

charent.cf

installprefs.acl

ixlang.cf

pubview.cf

pubview.fnt

tfmfont.cf

tfmscaling.cf

tfontsub.cf

wcharset.cf

wfontsub.cf

xcharset.cf

xfontsub.cf

The custom\lib directory also has a subdirectory to hold native shared libraries for platform-specific use:

- dll

  Holds Windows dynamic link libraries, or DLL files (.dll).

  The path to this directory is prepended to the system *PATH* environment variable.

The custom\lib directory can have an ixlang subdirectory, which holds a custom ixlang.cf file and index mapping files like those found in *Arbortext-path*\lib\ixlang.

- publishingrules subdirectory

  Holds publishing rules .prcf files which contain definitions of publishing rules and publishing rule sets.

- pubview subdirectory

  Holds pubview.cf and pubview.fnt files.

The default path is *Arbortext-path*\pubview. Then the *Arbortext-path*\custom\pubview directory is prepended to it.

- scripts subdirectory

  Holds .acl (Arbortext Command Language), .vbs (VBScript), and .js (JavaScript and JScript) files. Supports one level of subdirectories.

  The scripts in this directory can be called from scripts or applications in the custom\init directory, which is processed at startup time. Scripts placed here can be accessed using the source or require ACL commands. A customized menu item or button can call a script in custom\scripts when invoked.

  If there are any subdirectories of the custom\scripts directory, those subdirectories are prepended to the load path. Then the custom\scripts parent directory is prepended to the load path.

- stylermodules subdirectory

  Holds Arbortext Styler stylesheet modules. Any modules stored in this directory are automatically available to Arbortext Styler.

- tagtemplates subdirectory

  Holds .tpl files. You can also put custom tag templates you want associated with a particular document type into a custom\doctypes*doctype*\tagtemplates directory or in the original location of the document type's *doctype*\tagtemplates directory.

  If the user clicks the **New** button from the **Tag Templates** dialog box, Arbortext Editor will use the first directory with write access for that user in the tag template path.

  If the *APTTAGTPLDIR* environment variable is set, this path is prepended to it.

- init subdirectory

  Holds .acl, .js, .class, and .vbs files.

  The init subdirectory is processed last at startup time. All files of the supported application types are executed. No nested subdirectories of custom\init are supported. This directory is processed after the other *Arbortext-path*\custom subdirectories so that its scripts and applications can rely on paths already established during startup.

  If you are putting custom applications on the Arbortext PE server, use the init directory for your custom .acl, .js, .class files.

In the startup process, the `custom\init` directory is processed after `_main.acl` but before `arbortext.wcf`. See the online help topic *Startup command files* for complete startup processing information.

The supported application types are:

- `.acl` (Arbortext Command Language) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

- `.js` (JavaScript or JScript) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP clients. You need to specify the JavaScript interpreter engine to use in processing `.js` files. Refer to Specifying the JavaScript Interpreter Engine on page 31 for more information.

- `.class` (Java) files

  Java `.class` files in this directory must be compiled Java classes that are not part of a named package. You can also put a `.class` file in `custom\init` that calls into a `.jar` file located in the `custom\classes` directory.

  The Java class must also implement a `public static void main(String[] args)` method, which will be called with an empty string array. If the `.class` file does not implement this method, an error is reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

- `.vbs` (VBScript) files

  Errors are reported to Arbortext Editor.

- `editinit` subdirectory

  Holds `.acl`, `.js`, `.class`, and `.vbs` files. Note that when you run Arbortext Editor with the `-c` option, any applications in this subdirectory are not executed at startup.

  All files of the supported application types are executed each time a non-ASCII document is opened for editing. Files in this directory act on a document opened in the Edit window. File in this directory act on a document opened using ACL when the `0x8000` flag is used with the `doc_open` function. File in this directory act on a document opened using AOM when the `OPEN_EDITINIT` flag is used with the **Application.openDocument** method.

  The `editinit` subdirectory is processed before any document type command files, document type instance command files, and document command files.

The supported application types are:

- `.acl` (Arbortext Command Language) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.js` (JavaScript or JScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.class` (Java) files

  Java `.class` files in this directory must be compiled Java classes that are not part of a named package. The Java class must also implement a `public static void main(String[] args)` method, which is called with an empty string array. You can put a `.class` file in `custom\init` that calls into a `.jar` file located in the `custom\classes` directory. Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.vbs` (VBScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

Error Reporting for the custom\init Directory

Errors caused by mistakes in custom code in the *`Arbortext-path`*`\custom\init` directory are reported with both the error message and the name of the initialization file causing the error. Note the following:

- If Arbortext Editor is not running interactively (batch mode), no errors are reported and the errors are not logged.
- Arbortext Publishing Engine records errors and reports them to its HTTP clients in an HTML error page.
- ACL, JavaScript, and Java class errors are reported to the Arbortext Editor interface or held by Arbortext Publishing Engine to be sent to HTTP clients making requests.

Additional Information

If you are using the AOM, refer to the documentation for `Application.getCustomDirectory`. Refer to XUI Overview on page 114 for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following `set` command options and environment variables affect custom path search lists. They are documented in the online help.

`set catalogpath`

`set composerpath`

`set dialogspath`

`set ditapath`

`set entitypath`

`set framesetpath`

`set graphicspath`

`set javaclasspath`

`set libpath`

`set loadpath`

`set pdfconfigfile`

`set tagtemplatepath`

`set userdictpath`

Related Topics

- Using the custom directory for custom applications on page 24
- Description of the application directory structure on page 25
- Startup command files
- The following `set` command options and environment variables affect custom path search lists:
  - `set catalogpath`
  - `set composerpath`
  - `set dialogspath`
  - `set ditapath`
  - `set entitypath`
  - `set framesetpath`
  - `set graphicspath`
  - `set javaclasspath`
  - `set libpath`
  - `set loadpath`

- `set pdfconfigfile`
- `set tagtemplatepath`
- `set userdictpath`
- *APTTEXFONTS* environment variable

For information on creating and implementing custom applications, see the *Programmer's Reference* and the *Customizer's Guide*.

If you are using the AOM, refer to the documentation in the *Programmer's Reference* for `Application.getCustomDirectory`.

Refer to XUI Overview on page 114 for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

Using the Custom Directory for Custom Applications

The *Arbortext-path*`\custom` subdirectory structure provides the means to implement custom applications. Where your application should be placed depends on the application purpose and programming language.

If you're implementing custom applications or scripts, the following information will assist you in determining the approach and location for your files:

- A custom Java program can be placed in `custom\init`, which supports a `.class` file that must implement a `public static void main (String[] args)` method. The method will be called at startup with no arguments (an empty `String` array). If an error occurs, it's reported interactively for Arbortext Editor or sent to the HTTP client for the Arbortext Publishing Engine.

  A custom Java program can also be placed in `custom\classes`, which supports `.class` or `.jar` files.

  We recommend putting Java applications in the `custom\classes` directory and calling or initializing them from the `custom\init` directory.

  Paths to `.jar` files in `custom\classes` are automatically prepended to the embedded Arbortext Editor Java class path. Then the path to `custom\classes` is prepended, putting it first in the search order.
- A custom JavaScript, JScript, VBScript, or ACL application can be placed in `custom\init` or in `custom\scripts`. If you place your scripts in the `custom\scripts` directory, you can call them from a script or scripts you place in `custom\init` (which is processed at startup). Any code that exists outside a function definition in a script from `custom\init` is executed at

startup time. Errors are reported if running interactively, otherwise they're suppressed.

You can create a simple JavaScript example file called `simple_init.js`. The script should contain the following line:

```
Application.alert("Hello from JavaScript");
```

Put the `simple_init.js` file in *`Arbortext-path`*`\custom\init`.

When the startup process loads scripts from `custom\init`, you will see a dialog box showing the `Hello from JavaScript` message.

Description of the Application Directory Structure

The *`Arbortext-path`*`\application` subdirectory supports installing an application into the Arbortext Editor and Arbortext Publishing Engine install trees. Arbortext Editor and the Arbortext Publishing Engine automatically search for subdirectories of the `application` directory at startup.

`Arbortext-path``\application` must contain a uniquely named subdirectory for each distributed application. Arbortext recommends using the naming pattern for a unique qualified Java class name:

`com.company-name.application-name`

Each unique subdirectory of the `application` directory must also contain an `application.xml` configuration file which describes various aspects of the application, such as its release version and supported versions of Arbortext products. At startup, Arbortext Editor and the Arbortext Publishing Engine search the `application` directory for any subdirectories containing an `application.xml` configuration file. The `application.xml` file contents provide the criteria to determine whether the application should be loaded. The `application` directory must be located using a file system; HTTP references are not supported.

Subdirectory Structure

A subdirectory of the `application` directory can be structured the same as the `custom` directory to take advantage of automatic Arbortext Editor and Arbortext Publishing Engine startup processes. For example, if the uniquely named directory contains `graphics` or `entities` directories, those directories are automatically added to the search paths constructed at startup.

An application path could be something like:

`application\`*`com.company-name.application-name`*

Refer to the Description of the custom directory structure on page 13 for the names and descriptions of each supported subdirectory.

Note

When Arbortext Editor or the Arbortext Publishing Engine constructs search paths, subdirectories of the `custom` directory take precedence over any corresponding subdirectories under the `application` directory. When search lists are constructed at startup, the first path in any search list will be the appropriate `custom` directory followed by any applicable directory under the `application` directory. For example, in constructing the graphics search path list at startup, `custom\graphics` would precede `application\com.arbortext.sample\graphics`. An `application\graphics` directory with no `application.xml` file will be ignored during startup.

When implementing a custom application using the `application` directory structure, you can add supplemental directories as needed to support your application. However, your application code must be aware of these directories and how to use them.

Application Startup File

The *`Arbortext-path`*`\doctypes\appcfg\application.xml` file provides a basic template for defining information about the custom application. You can make a copy of `doctypes\appcfg\application.xml` to use as a template to create the file that will eventually be distributed with the application. The `application.xml` file must be placed in the application's top level directory, for example:
`Arbortext-path``\application\com.`*`company.application-package-name`*`\application.xml`

In the template `application.xml` file, you can specify a list of elements that describe the application. If the custom application determines its criteria is not met and the application is not to be loaded, then these values are ignored. The base element for the file is the `ApplicationConfiguration` element. This element has a required attribute called *installType* that determines the type of Arbortext Editor installation for which this application is supported. The supported value is `full` meaning the application is only supported in the full installation of Arbortext Editor. The value `any` previously indicated whether compact installation of Arbortext Editor was supported .

The following other elements are supported in the `application.xml` file:

- `Name` (required)
- `Description`
- `LicenseNumber` is only for an application distributed by Arbortext
- `Version` (required)

- `Date`
- `Copyright`
- `Vendor`
- `RequiredApplications` is for other applications that are required for this application to run correctly. You must enter the qualified name for the application in the *qualifiedName* attribute and a human-readable name in the *name* attribute.
- `SupportedProducts`

  A `Product` element has attributes for specifying the name (required), minimum version (required), and maximum version of the Arbortext product that supports the custom application or application. The `Product` specification helps the launching Arbortext product determine whether it should load this custom application by matching criteria specified in this section.

  The name must be one or more of the following:

  - Arbortext Editor
  - Arbortext Publishing Engine
  - Arbortext Architect
  - Arbortext Editor with Styler

  The version must follow the convention used by Arbortext products, such as 5.2, 5.2 M040, or 5.3.
- `SupportedPlatforms`

  The section is reserved for future use. Windows is currently the only supported platform.
- `GlobalParameters`

  `Parameter` contains `ParameterName` and `ParameterValue` elements for specifying any global variables that the application may need when it's launched.

Related Topics

If you are using ACL, refer to the following ACL function descriptions:

- `application_name function`
- `get_custom_dir function`
- `get_custom_property function`

- `get_user_property function`
- `set_user_property function`

If you are using the AOM, refer to the documentation for `Application.getCustomDirectory`. Refer to XUI Overview on page 114 for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following attributes from the **Application** interface are also useful:

- `haveWindows`
- `initDone`
- `isE3`
- `customProperties`
- `userProperties`
- `name`

Using the Application Directory for Custom Applications

The *`Arbortext-path`*`\application` subdirectory provides the means to implement a custom application that uses a special configuration file to determine whether it should be loaded at startup. The `application` directory uses the same principles of structure as the `custom` directory.

The *`Arbortext-path`*`\application` directory is processed at startup. If you add a custom application after startup, you must exit and restart Arbortext Editor or stop and restart the Arbortext Publishing Engine to have it recognized. You also have the option to issue the `f=init` function to re-initialize the Arbortext PE sub-processes. Refer to *Configuration Guide for Arbortext Publishing Engine* for more information.

Rules for using the `application` directory are:

- Your custom application must be contained in a uniquely named subdirectory of the `application` directory.
- You must have an `application.xml` configuration file in the uniquely named subdirectory that sets the conditions for loading the application.
- The same set of subdirectories supported by the `custom` directory are supported for the uniquely named subdirectory of the `application`

directory. At startup, the supported directories are automatically detected and used in constructing search paths.

- Any other subdirectory of the `application` directory will be ignored at startup. For example, an `application\graphics` subdirectory with no `application.xml` file will be ignored during startup.

Arbortext has developed proprietary custom applications that are deployed using the `application` subdirectory structure. A uniquely named subdirectory contains all the necessary components to run an application within Arbortext Editor as well as the Arbortext Publishing Engine.

The following information will help determine an approach for a custom application.

- You can have additional subdirectories for your custom application. You are not limited to the subdirectories supported by the `custom` directory. However, these additional directories are not automatically recognized during the startup process.
- Processing each unique application's subdirectories follows the same rules for processing `custom` subdirectories. Recall that the application's subdirectories come after the `custom` subdirectories in constructing any applicable search paths for the session.
- If you decide not to use a particular supported subdirectory, you can improve performance by omitting the directory to reduce the length of a search path that would contain it.
- You can use the *APTAPPLICATION* environment variable to set the path to one or more `application` directories.
- An application should not write data to its own application directory. An application user may not have write permission access to this application directory, for example, any `C:\Program Files` directories on Windows (the location where Arbortext Editor and the Arbortext Publishing Engine are typically installed).

Deploying Zipped Customizations

You can deploy not only `custom` directories, but also `application` and content management system adapters directories in a compressed zip file. Using a zip file to distribute your customizations has the following advantages:

- You can host your customizations on a web server.

In this case, use the HTTP or HTTPS URL to the zip file as the value for the *APTCUSTOM* environment variable.

- Your customizations will be available to users when they cannot access your network.

  If you use a shared network folder to host your customizations, users do not have access to those customizations when the network is unavailable. If you use a zip file to distribute your customizations, Arbortext Editor unzips those customizations to a directory in the Arbortext Editor cache directory (`.aptcache\zc`). At start up, Arbortext Editor checks to see whether the zip file has been updated. If it has, Arbortext Editor downloads and uncompresses the updated customizations. If not, Arbortext Editor continues to use the customizations stored in the local cache. If the network is unavailable to a user, your customizations are still available to that user in the local cache. Note that the user must also have a fixed Arbortext Editor license on their system to work away from the network.

- Network traffic might be reduced.

  Since the zip file containing your customizations is only downloaded once over the network, and then only if it has been updated, traffic on your network might be reduced. If you store your unzipped customizations in a shared network folder, Arbortext Editor might have to access that folder several times over the course of a session.

- Customizations stored in a compressed zip file are harder to change accidentally than customizations stored in a directory structure.

Note that you cannot use a zip file to distribute a customized `installprefs.acl` in the `custom\lib` directory. You can use the *APTINSTALLPREFS* environment variable to specify the location of a custom `installprefs.acl` file.

Note also that you cannot include the following font configuration files in the `lib` subdirectory of a zipped `custom` directory:

- `charent.cf`
- `wcharent.cf`
- `wfontsub.cf`
- `charmap.cf`

These files are processed before a zipped `custom` directory when Arbortext Editor starts up, so the files cannot be processed when deployed in that way.

Specifying the JavaScript Interpreter Engine

Both JavaScript and JScript files have a `.js` file extension. By default, Arbortext Editor and the Arbortext Publishing Engine interpret `.js` files as Rhino JavaScript files. You should specify the JavaScript interpreter for a JavaScript or JScript `.js` file. This is especially important if you have `.js` files of both types.

We recommend adding a comment line to your script that specifies either the Rhino JavaScript engine (the default) or the Microsoft JScript engine as shown in the following examples. The first line of your `.js` file must be a comment starting with `//`.

To specify the Rhino JavaScript interpreter:

```
// type="text/javascript"
```

To specify the Microsoft JScript interpreter:

```
// type="application/jscript"
```

The specification can be enclosed in a script tag. Both of the following examples are a valid specification for JScript:

```
// <script type="application/jscript">
// type="application/jscript"
```

You can also specify the JavaScript interpreter using the ACL `set javascriptinterpreter` command. You can specify it in an ACL file placed in the *`Arbortext-path`*`\custom\init` directory, where it will be processed at startup. For information on setting the interpreter using ACL, see the online help topic for `set javascriptinterpreter`.

2

Customizing Your Site's Profiling Configuration

Customizing Your Site's Profiling Configuration

Profiling sections of documents let you designate that certain sections contain information targeted at a specific audience or contain information that only applies when a particular set of circumstances exists. This chapter describes how to configure profiling specific to your site’s needs.

Profiling Overview

Profiling is a means to provide specific content for a selected audience or for a specific application. Using profiling, authors can include all document variations in one file, and use profiles to control what elements appear in published versions of a document. By comparing the selected audience with each element's audience profile, Arbortext Editor strips out irrelevant content and assembles a custom publication.

Individual profiles specify that content can have one or more than one profile of a particular class. Classes may contain standard and unique profiles.

- Standard individual profiles apply one or more profiles in a class to an element.
- Unique individual profiles apply one and only one profile in a class to an element.

Two types of profile groups exist. **Apply profile groups** specify a collection of individual profiles defined as a named profile group an author can apply to an element in a single step. **Set profile groups** specify a collection of individual profiles an author can choose at publishing time in a single step.

Authors apply profile values to elements at editing time by setting certain element attributes to specific values as defined in profile configuration (`.pcf`) files. Individual document types reference the `.pcf` file containing the profiling definitions defined for the document type. Multiple document types can reference a single `.pcf` file.

You can configure colored shading to differentiate between profile, profile groups, or individual values. Refer to Using shading for profiled elements for further information.

.pcf (Profile Configuration File)

A profile configuration file (`.pcf`) is an XML document specifying profile values that can be applied to any elements (or a limited number of elements) in a document type. A document type's `.dcf` file specifies the `.pcf` file to use for the document type's profiling configuration. Several document types can use the same `.pcf` file for their profiling configurations.

A `.pcf` file has the following structure:

- A top-level `<Profiles>` element contains all of the `<ProfileClasses>` elements in the configuration file.
- `<ProfileClasses>` elements define the profile classes of related individual profiles and groups of individual profiles for applying at editing time and setting at publishing time. `<ProfileClasses>` elements contain the `<Profile>`, `<ApplyProfileGroup>`, and `<SetProfileGroup>` elements.

  If you are working with profile shading, you can set a value for the **conflictShadingBackground** attribute for a `<ProfileClasses>` element to provide a conflict color. This color will be applied to an element in a document if it has been assigned multiple profile values, each configured with different shading colors.

  > **Note**
  >
  > Although it is possible to specify a conflict color on any `<ProfileClasses>` element in the profile configuration file (`.pcf`), the color must be defined on the first `<ProfileClasses>` element to be effective.

- `<Profile>` elements define individual profiles that authors can apply to elements at editing time and select to produce profiled output. With the `<Profile>` element, you can specify:
  - The specific elements to which the profile is restricted
  - The specific elements from which the profile is restricted
  - Sub classes, or folders, of related profiles
  - Allowed values for the profile
  - That the profile is restricted to one and only one profile value
  - Shading colors

Set the **shadingBackground** attribute for the whole profile (`Profile` element), a profile sub-category (`ProfileFolder` child element), or a profile's individual values (`Allowed` child element)

- `<ApplyProfileGroup>` elements specify groupings of individual profiles the author can apply to elements at editing time.
- `<SetProfileGroup>` elements specify groupings of individual profiles the author can set at publishing time. Individual profiles can be included and excluded using logical expressions.

Configuring Profiles

This section covers the profile configuration process and provides examples of profile configurations.

Configuration Process

Before actually configuring your profiling, determine the proper profiles to create for your site. Consider the following items:

- Create profiles so that your biggest possible audience does not require that any profiles are applied. This will cut down on the time needed to profile a document.
- Determine whether it will be more work to profile a document to include elements or exclude elements. It may create less work for authors if you create a `NOT Model 123` profile instead of a `Model 123` profile.
- Avoid creating profiles that are subsets of one another. For example, in the `Security` profile class, do not create a general `Employees` profile and specific `Managers` and `Trainees` profiles. This may cause problems for those applying the profiles. Instead of creating the general `Employees` profile, create the `Managers` and `Trainees` profiles.

Use the following procedures to create or update a profiling configuration file.

Specifying the .pcf File to Use

1. Using Arbortext Architect or Arbortext Editor, open the `.dcf` file of the document type for which you want to configure profiles.
2. Locate the `Profiling` element. If the file doesn't include a `Profiling` element, add one.
3. Place your cursor next to the `Profiling` element and choose **Edit ▸ Modify Attributes**. Enter the name of the `.pcf` file containing the profiling configuration you want to use with this document type and choose **OK**. (If the

`.pcf` file is not in the same directory as the `.dcf` file, enter the full path and file name of the `.pcf` file.)

4. Save the document and close Arbortext Architect or Arbortext Editor.

Configuring the Profiles

1. Using Arbortext Architect or Arbortext Editor, open the `.pcf` file in which you want to configure profiles.
2. If this is a new (empty) `.pcf` file, add the top-level `Profiles` element.
3. Profiles are categorized within `ProfileClasses` elements that define the profiles elements can have. Create a new `ProfileClasses` element. A child `Profile` element is automatically created and you are prompted to edit the `Profile` element's attributes. (You must have the **Edit ▸ Force Required Attributes Entry** preference selected for the **Modify Attributes** dialog box to open automatically.)
4. Type a descriptive name for the attribute in the **alias** field. This is the profile name that will appear in Arbortext Editor profiling dialog boxes for assigning profiles and publishing documents.
5. Type a valid attribute name in the **attribute** field. You must specify a common attribute that can appear on every element in the DTD, and it must have a declared value of CDATA.
6. Choose **OK** to create the profile.
7. Define the allowed values this profile can have by placing the cursor next to the `Profile` element and inserting a child `Allowed` element and `value` attribute for each possible value.For example, if you specified the `os` attribute of the `Profile` element, you might want to specify values of `Windows` and `UNIX` for the `value` attributes of two `Allowed` elements.
8. Repeat the previous steps to configure additional profile classes.
9. When you've completed adding profiles to the configuration file, save and close the `.pcf` file.

Configuring Profile Shading

1. Using Arbortext Architect or Arbortext Editor, open the `.pcf` file in which you want to configure profiles.
2. Locate the `Profile` element for which you wish to set shading. You can choose to set a shading color for the whole profile (`Profile` element), a profile sub-category (`ProfileFolder` child element), or for a profile's individual values (`Allowed` child element).
3. Place your cursor next to the element for which you wish to set shading, and choose **Edit ▸ Modify Attributes**.

4. In the **Modify Attributes** dialog box, select a color value for the **shadingBackground** attribute.
5. Choose **OK** to exit the dialog box.
6. Locate the `ProfileClasses` element for which you wish to provide a conflict color.This color will be applied to an element in a document if it has been assigned two profile values, each configured with different shading colors.

> **Note**
>
> Although it is possible to specify a conflict color on any `<ProfileClasses>` element in the profile configuration file (`.pcf`), the color must be defined on the first `<ProfileClasses>` element to be effective.

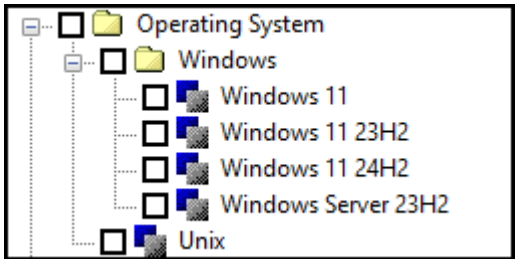
7. Choose **Edit ▸ Modify Attributes**. In the **Modify Attributes** dialog box, select a color value for the **conflictShadingBackground** attribute.
8. Choose **OK** to exit the dialog box.
9. When you've finished configuring profile shading, save and close the `.pcf` file.

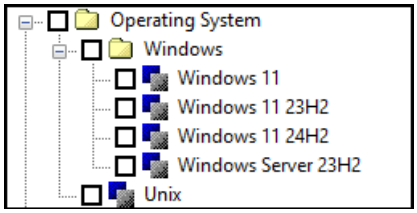
Profiling Configuration Examples

A sample `.pcf` file accompanies the sample axdocbook template. The `.pcf` file is stored at *`Arbortext-path`*`\doctypes\axdocbook\axdocbook.pcf`. Several of the following examples are included in `axdocbook.pcf`.

Nesting Profiles

Profile classes can contain folders containing more folders and profiles. Using such a structure provides a categorization of related profiles. In this example, several Windows platforms are categorized in a parent Windows folder.





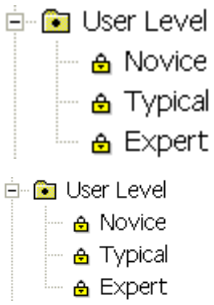
Example of nested profiles in the **Apply Profiles** dialog box.

This profiling configuration is created with the following markup:

```
<Profile attribute="os" alias="Operating System">
 <ProfileFolder name="Windows">
 <Allowed value="Windows 11"/>
 <Allowed value="Windows 10"/>
 <Allowed value="Windows Server 23H2"/>
 </ProfileFolder>
 <Allowed value="Unix"/>
</Profile>
```

Restricting Profiles to or from Specific Elements

By restricting profiling to only certain elements, you can ensure that information is always included or included in only certain circumstances. In this example, if the `toc` element has a `role` attribute set to the value `required`, the element cannot be profiled by the users level of expertise. This ensures that the Table of Contents is always included when the document is published. If the user attempts to profile a `toc` element with its `role` attribute set to the value `required`, the profiles will be unavailable:



Example of restricted profile values.

This profiling configuration is created with the following markup:

```
<Profile attribute="userlevel" alias="User Level">
 <NotProfileElement element="toc">
 <AttributeTest name="role" value="required"/>
 </NotProfileElement>
 <RadioChoice>
 <Allowed value="Novice"/>
 <Allowed value="Typical"/>
 <Allowed value="Expert"/>
 </RadioChoice>
```

```
</Profile>
```

Using Logical Expressions when Configuring Profiles

Set profile groups define a collection of individual profiles an author can choose at publishing time in a single step. When creating set profile groups, you can use logical expressions to specify publishing that is dependent on profile value relationships. When using logical expressions, ensure that the names assigned to the SetProfileGroup elements clearly communicate to the user the profiles that will be published.

In this example, different user levels are grouped. Elements profiled for all of the user levels represented by the selected group will be published.

Example of set profile groups.

This profiling configuration is created with the following markup:

```
<SetProfileGroup name="Novice and Typical User Levels">
 <LogicalExpression>
 <LogicalGroup operator="OR">
 <ProfileRef alias="User Level" value="Novice"/>
 <ProfileRef alias="User Level" value="Typical"/>
 </LogicalGroup>
 </LogicalExpression>
</SetProfileGroup>
<SetProfileGroup name="Expert and Typical User Levels">
 <LogicalExpression>
 <LogicalGroup operator="OR">
 <ProfileRef alias="User Level" value="Expert"/>
 <ProfileRef alias="User Level" value="Typical"/>
 </LogicalGroup>
 </LogicalExpression>
</SetProfileGroup>
<SetProfileGroup name="Typical Windows Customer">
 <LogicalExpression>
 <LogicalGroup operator="AND">
 <ProfileRef alias="User Level" value="Typical"/>
 <LogicalGroup operator="OR">
 <ProfileRef alias="Operating System" value="Windows 11"/>
 <ProfileRef alias="Operating System" value="Windows 10"/>
```

```
 <ProfileRef alias="Operating System" value="Windows Server 23H2"/>
 </LogicalGroup>
 <ProfileRef alias="Security Level" value="Customer"/>
 </LogicalGroup>
 </LogicalExpression>
</SetProfileGroup>
```

Profiling API

Profiles are organized into folders and sub-folders containing one or more profile values. You can visualize the resulting structure as a tree of information, with each profile (along with its folders, sub-folders, and allowed values) being a branch of the tree. Each folder, sub-folder, or value is considered to be a `node` (or `profilenode`) on the tree. The profile tree is a hierarchical structure containing `profilenode` objects and branches that link the different `profilenodes` with each other. Each branch has a top-level root node with sub-folder nodes (if any) and value nodes that represent the leaves of the tree.

The profiling API consists of ACL functions that walk a profile tree and traverse the profilenodes to determine the following information:

- The different properties of each `profilenode`. (For example, to determine the type of node the `profilenode`is.
- Relative location information about the `profilenode` such as the node's ancestors, children, and so on.

A `profilenode` can be one of the following types:

- STANDARD_PROFILE, RADIO_PROFILE or FOLDERED_PROFILE — The type assigned to the top-level (root) `profilenode`.

  A profile is of type RADIO_PROFILE if takes on radio choices as allowed values (unique profiles). A profile is of type FOLDERED_PROFILE if it contains folders. Otherwise, the profile is of type STANDARD_PROFILE.
- PROFILE_FOLDER
- ALLOWED_VALUE

The following elements in a profile configuration file are assigned a `profilenode` value as defined in the following table:

Types of Profilenodes

Element	Profilenode Type and Value
`<Profile>`	• STANDARD_PROFILE = `1` • RADIO_PROFILE = `2` • FOLDERED_PROFILE = `3`
`<ProfileFolder>`	PROFILE_FOLDER = `4`
`<Allowed>`	ALLOWED_VALUE = `5`
Unrecognized markup	INVALID_PROFILE = `0`

The following ACL functions support profiling and allow for site-specific customizations of Arbortext Editor profiling capabilities. Refer to the Arbortext Editor online help for detailed descriptions of each function.

Profilenode Functions

`profilenode_ancestors`(*`profilenode`*, *`arr`*)

Returns the number of folders that are ancestors of the node identified by the specified node.

`profilenode_attr` (*`profilenode`*)

Returns the name of the profile attribute for the specified node.

`profilenode_children_nodes`(*`profilenode`*, *`arr`*)

Returns the number of nodes that are children of the specified node.

`profilenode_default_value`(*`profilenode`*)

Returns the default value for a RADIO_PROFILE node as specified in the profile configuration file.

`profilenode_default_value_node`(*`profilenode`*)

Returns the default value profilenode for a RADIO_PROFILE node as specified in the profile configuration file.

`profilenode_element_allowed`(*`profilenode`*, *`tagname`*)

Returns `1` if the element can be profiled using the profilenode.

`profilenode_element_attr_tests`(*`profilenode`*, *`tagname`*, *`arr`*)

Returns the number of attribute name(s) and value(s) for an element that a particular profile could be applied to or not applied to.

`profilenode_elements_list`(*`profilenode`*, *`arr`*, *`not_indicator`*)

Returns the number of elements that a particular profile could be applied to or not applied to.

profilenode_is_foldered(*profilenode*)

Returns 1 if the root node of the specified node is a FOLDERED_PROFILE node.

profilenode_is_radiochoice(*profilenode*)

Returns 1 if the root node of the specified node is a RADIO_PROFILE node.

profilenode_is_standard(*profilenode*)

Returns 1 if the root node of the specified node is a STANDARD_PROFILE node.

profilenode_name (*profilenode*)

Returns the profile alias, folder name, or profile value of a profilenode.

profilenode_parent (*profilenode*)

Returns the *profilenode* of the immediate ancestor of the specified node.

profilenode_rootnode (*profilenode*)

Returns the top-level or root node for that profile class.

profilenode_shadingbackground(*profilenode*)

Returns the shading color for the specified profile node.

profilenode_type (*profilenode*)

Identifies a profilenode's TYPE.

profilenode_valid (*profilenode*)

Returns 1 if the specified node is a valid profilenode identifier.

profilenode_value_nodes(*profilenode*, *arr*)

Returns the number of value profilenodes for the specified node.

profilenode_value_separator(*profilenode*)

Returns the value separator corresponding to the specified node. The default value is a semicolon.

profilenode_values (*profilenode*, *arr*)

Returns the number of allowed values for the specified node.

Profile Functions

profile_alias (*attr*[, *doc*])

Returns the alias name for the specified profile attribute.

profile_aliases (*arr*[, *doc*])

Returns the number of profile aliases defined in the current (or other) profiling configuration file.

profile_allowed (*alias*, *oid*)

Returns 1 if the specified element can be profiled using the specified profile. Returns 0 if the element cannot be profiled, or if either *oid* or *alias* is invalid.

profile_attr (*alias*[, *doc*])

Returns the name of the profile attribute for the specified profile.

profile_attrs (*arr*[, *doc*])

Returns the number of profile attributes defined in the configuration file associated with the profiling session.

profile_config ([*doc*])

Returns a document identifier for the current (in-memory) profiling configuration file.

profile_conflictshadingbackground([*doc*])

Returns the conflict shading color for the profile for the specified document.

profile_default_value(*alias*[, *doc*])

Returns the default value for the specified RADIO_PROFILE.

profile_default_value_node(*alias*[, *doc*])

Returns the default value profilenode for the specified RADIO_PROFILE.

profile_element_allowed(*alias*, *tagname*[, *doc*])

Returns 1 if the specified element can be profiled using the specified profile.

profile_element_attr_tests(*alias*, *tagname*, *arr*[, *doc*])

Returns the number of attribute name(s) and value(s) for the specified element that the specified profile can be applied to or not applied to.

profile_elements_list(*alias*, *arr*, *not_indicator*[, *doc*])

Returns the number of elements the specified profile could be applied to or not applied to.

profile_is_foldered (*alias*[, *doc*])

Returns 1 if the specified profile class is a FOLDERED_PROFILE.

profile_is_radiochoice(*alias*[, *doc*])

Returns 1 if the specified profile class is a RADIO_PROFILE.

profile_is_standard (*alias*[, *doc*])

Returns 1 if the specified profile class is a STANDARD_PROFILE.

profile_resolution (*oid*, *logical_expression*)

Returns 1 if the specified element would be included if the profile was resolved using the specified logical expression.

profile_rootnode (*alias*[, *doc*])

Returns the *profilenode* of type STANDARD_PROFILE, RADIO_PROFILE, or FOLDERED_PROFILE of a profile class.

profile_rootnodes (*arr*[, *doc*])

Returns the number of profilenodes of type STANDARD_PROFILE, RADIO_PROFILE or FOLDERED_PROFILE in the current (or other) profile configuration file.

profile_shadingbackground(*alias*[, *doc*])

Returns the shading color of the profile attribute for the specified alias..

profile_type (*alias*[, *doc*])

Returns an integer identifying the profile type of the specified alias.

profile_valid (*alias*[, *doc*])

Returns 1 if the specified alias is a valid profile.

profile_value_node (*alias*, *value*[, *doc*])

Returns the allowed value profilenode for the specified value in the profile class identified by the specified alias.

profile_value_nodes (*alias*, *arr*[, *doc*])

Returns the number of allowed value profilenodes for the profile class identified by the specified alias.

profile_value_separator(*alias*[, *doc*])

Returns the value separator for the specified profile. The default value is a semicolon.

profile_values (*alias*, *arr*[, *doc*])

Returns the number of allowed values for the profile class identified by the specified alias.

profile_values_shadingbackground(*alias*, *arr*[, *doc*[, *IncludeProfileElement*]])

Returns the colors for the profile attribute for the specified alias.

Profile Group Functions

apply_profile_group (*apply_profile_group_name*, *arr*[, *doc*])

Returns the number of profile classes that are included in the specified apply profile group.

apply_profile_group_allowed(*apply_profile_group_name*, *oid*, *arr[]*)

Returns 1 if the profile group *apply_profile_group_name*, is allowed on the element *oid*.

apply_profile_group_value_nodes(*apply_profile_group_name*, *arr*[, *doc*])

Returns 1 if the specified profile group is allowed on the specified element.

apply_profile_groups (*arr*[, *doc*])

Returns the number of apply profile groups specified in the profile configuration file.

set_profile_group (*set_profile_group_name*[, *doc*])

Returns the profile configuration file markup for the specified set profile group

set_profile_groups (*arr*[, *doc*])

Returns the number of set profile groups specified in the profile configuration file.

set_profile_groups_expressions(*arr*[, *doc*])

Returns the number of set profile groups, and the profile classes and relationships between profile values for the corresponding resolution group specified in the profiling configuration file.

Profiling DTD Element Reference

The location of the profiling document type definition is *Arbortext-path*\doctypes\profiling\profiling.dtd.

Allowed Element

Synopsis

Mixed content model:

```
Allowed
Empty
```

Attributes:

```
value CDATA #REQUIRED
```

Description

The <Allowed> element specifies the only allowed value for a profile.

The element has no child elements.

The <Allowed> element has the following attribute:

- *value = CDATA*

  The allowed value for the parent profile.

ApplyProfileGroup Element

Synopsis

Mixed content model:

```
ApplyProfileGroup
(ProfileRef)+
```

Attributes:

```
name CDATA #REQUIRED
```

Description

The <ApplyProfileGroup> element specifies a named apply profile group.

The element can have the following child element:

`<ProfileRef>`

The `<ApplyProfileGroup>` element has the following attribute:

- *name* = *CDATA*

  Specifies the name of the profile group.

AttributeTest Element

Synopsis

Mixed content model:

```
AttributeTest
EMPTY
```

Attributes:

```
name CDATA #REQUIRED
value CDATA #IMPLIED
```

Description

The `<AttributeTest>` element specifies whether an attribute test must be performed.

The element has no child elements.

The `<AttributeTest>` element has the following attributes:

- *name* = *CDATA*

  Specifies the attribute name to test.

- *value* = *CDATA*

  Specifies the value to test for. If *value* is not specified, and the tested attribute has any declared value, the test will return `TRUE`. If *value* is set to `ATI#UNDECLARED`, the test will return `TRUE` only if the test attribute is undefined.

LogicalExpression Element

Synopsis

Mixed content model:

```
LogicalExpression
(LogicalGroup | LogicalNOT)
```

Attributes:

```
None
```

Description

The <LogicalExpression> element specifies a logical expression to use in a set profile group.

The element can have the following child elements:

<LogicalGroup>, <LogicalNOT>

The <LogicalExpression> element has no attributes.

LogicalGroup Element

Synopsis

Mixed content model:

```
LogicalGroup
((ProfileRef | LogicalGroup | LogicalNOT),
 (ProfileRef | LogicalGroup | LogicalNOT)+)
```

Attributes:

```
operator (AND | OR | XOR | EQUAL) #REQUIRED
```

Description

The <SetProfileGroup> element defines a logical expression group

The element can have the following child elements:

<ProfileRef>, <LogicalGroup>, <LogicalNOT>

The <LogicalGroup> element has the following attribute:

- *operator* = AND | OR | XOR | EQUAL

  Specifies the logical operator to use. The operators have the following resolutions when comparing values A and B:

  - AND — A logical conjunction. The expression is true if both A and B are true.
  - OR — A logical disjunction (an inclusive OR). The expression is true if A, B, or both, are true.
  - XOR — A logical inequivalence (an exclusive OR). The expression is true if either A or B is true, but false if both A and B are true.

- EQUAL — A logical equivalence. The expression is true if both A and B are true, or if both are false.

LogicalNOT Element

Synopsis

Mixed content model:

```
LogicalNOT
(ProfileRef| LogicalGroup)
```

Attributes:

```
None
```

Description

The <LogicalNOT> element specifies logical negation of an expression. That is, for value A, the expression is true if A is false and the expression false if A is true.

The element can have the following child elements:

<ProfileRef>, <LogicalGroup>

The <LogicalNOT> element has no attributes.

NotProfileElement Element

Synopsis

Mixed content model:

```
NotProfileElement
(AttributeTest*)
```

Attributes:

```
element NMTOKEN #REQUIRED
```

Description

The <NotProfileElement> element defines the elements to be restricted from having a particular profile.

The element can have the following child element:

<AttributeTest>

The <NotProfileElement> element has the following attribute:

- *element = NMTOKEN*

Specifies the name of the element from which the profile is restricted.

Profile Element

Synopsis

Mixed content model:

```
Profile
((ProfileElement* | NotProfileElement*),
 ((ProfileFolder | Allowed)+ | RadioChoice))
```

Attributes:

```
attribute NMTOKEN #REQUIRED
alias CDATA #REQUIRED
valueSeparator CDATA ";"
```

Description

The `<Profile>` element defines the profiles that are available to apply to an element.

The element can have the following child elements:

`<ProfileElement>`, `<NotProfileElement>`, `<ProfileFolder>`

The `<Profile>` element has the following attributes:

- *attribute = NMTOKEN*

  Defines the attribute in which to store the profile values. This can be an attribute value defined in the document type or a namespaced attribute value.

- *alias = CDATA*

  Specifies the name of the profile.

- *valueSeparator = CDATA*

  Specifies the delimiter used to separate multiple profile values specified on a particular attribute. The default value is a semicolon (`;`).

ProfileClasses Element

Synopsis

Mixed content model:

```
ProfileClasses
((Profile+, ApplyProfileGroup*, SetProfileGroup*))
```

Attributes:

```
none
```

Description

The <ProfileClasses> element defines the profiles that are available to apply to an element.

The element can have the following child elements:

<Profile>, <ApplyProfileGroup>, <SetProfileGroup>

The <ProfileClasses> element has no attributes.

- *authorModifiable* = true | false

  Specifies whether the author is allowed to modify the profile during an editing session.

ProfileElement Element

Synopsis

Mixed content model:

```
ProfileElement
(AttributeTest*)
```

Attributes:

```
element NMTOKEN #REQUIRED
```

Description

The <ProfileElement> element defines the elements to which the profile is restricted.

The element can have the following child element:

<AttributeTest>

The <ProfileElement> element has the following attribute:

- *element* = *NMTOKEN*

  Specifies the name of the element to which the profile is restricted.

ProfileFolder Element

Synopsis

Mixed content model:

```
ProfileFolder
(ProfileFolder+ | Allowed+)
```

Attributes:

```
name CDATA #REQUIRED
```

Description

The <ProfileFolder> element specifies the folder structure of a hierarchical (foldered) profile. Folders can contain folders.

The element can have the following child elements:

<ProfileFolder>, <Allowed>

The <ProfileFolder> element has the following attribute:

- *name = CDATA*

  Specifies the name of the folder.

ProfileRef Element

Synopsis

Mixed content model:

```
ProfileRef
Empty
```

Attributes:

```
alias CDATA #REQUIRED
value CDATA #REQUIRED
```

Description

The <ProfileRef> element specifies the profile to use in a group.

The element has no child elements.

The <ProfileRef> element has the following attributes:

- *alias = CDATA*

  The alias name associated with the profile being referenced.
- *value = CDATA*

  The allowed value associated with the profile being referenced.

Profiles Element

Synopsis

Mixed content model:

```
Profiles
(ProfileClasses+)
```

Attributes:

```
none
```

Description

The <Profiles> element is a the top-level element of the .pcf file.

The element can have the following child element:

<ProfileClasses>

The <Profiles> element has no attributes.

RadioChoice Element

Synopsis

Mixed content model:

```
RadioChoice
(Allowed*)
```

Attributes:

```
None
```

Description

The <RadioChoice> element specifies that a profile can only accept one value from a give list of values.

The element can have the following child element:

<Allowed>

The <RadioChoice> element has no attributes.

SetProfileGroup Element

Synopsis

Mixed content model:

```
SetProfileGroup
(ProfileRef | LogicalExpression)
```

Attributes:

```
name CDATA #REQUIRED
```

Description

The <SetProfileGroup> element specifies a combination of profile settings to be used during publishing or resolution.

The element can have the following child elements:

<ProfileRef>, <LogicalExpression>

The <SetProfileGroup> element has the following attribute:

- *name = CDATA*

  Specifies the name of the set profile group.

3

Customizing Help

Customizing Tag Help

Tag help is the help that appears when you place the mouse pointer over a tag in your document and press **SHIFT**+**F1** or **Help**. Tag help describes the elements declared in the document type that is being used.

Location of Tag Help Files

Tag help files are stored in a `\help` subdirectory of a document type's directory in *`Arbortext-path`*`\doctypes`. *`Arbortext-path`* is the directory where Arbortext Editor is installed. The `doctypes` directory in *`Arbortext-path`* in turn contains directories for the individual distributed document types. Each of those directories has a `\help` subdirectory.

As an illustration, if Arbortext Editor resides in the directory `c:\Program Files\Arbortext\Editor`, tag help for a document type would reside in `c:\Program Files\Arbortext\Editor\doctypes\`*`dtddir`*`\help`, where *`dtddir`* is the name of the directory containing a particular document type.

More specifically, if your document type is a customized document type installed in the directory `c:\apps\doctypes\mydoc`, the document type's tag help files will be stored in `c:\apps\doctypes\mydoc\help`.

Tag Help File Types

To create and modify help for element tags using Arbortext Editor, create and save the text for each tag in a separate file, typically using the same document type as that which the help supports. If the document type does not easily support text elements, PTC recommends using a document type such as XHTML.

For example, create help files supporting the memo SGML document type using the memo SGML document type. Create help files supporting the ATI XML DocBook document type using the ATI XML DocBook document type.

Creating Tag Help for a New Document Type

Use the following steps for creating tag help for a new document type.

1. Create a new document using the same document type as you are documenting. If the document type does not easily support text elements, use a document type such as XHTML.
2. Author your help text for a given tag. Use a separate file for each individual tag's help.
3. Save the document to the help subdirectory for your document type with the tag name as the file name and the appropriate extension for the document type.

For example, if you author a help file for an element `body` using the memo document type, save it in a file named `body.sgm`. If you author a help file for an element `sect1` using the ATI XML DocBook document type, save it in a file named `sect1.xml`

Save the file in the `help` subdirectory of *doctypedir*, where *doctypedir* is the directory where the document type resides.

4. Test the help file for a tag in a document based on that document type. Place the mouse pointer over the tag for which you've created help, and press **SHIFT**+**F1**. The new tag help is displayed.

Customizing Tag Help for an Existing Document Type

Use the following procedure to customize tag help for an existing document type, such as a document type delivered with Arbortext Editor.

1. Set the environment variable *APTHELPPATH* to specify a directory in which you will store the custom help files.
2. Copy the existing tag help files to this new directory.
3. Use Arbortext Editor to update the copies of the files as necessary.
4. Test your customized help by placing the mouse pointer over a tag and pressing **SHIFT**+**F1**. The customized tag help is displayed.

4

Customizing PDF Publishing

PDF Publishing Overview

There are several ways in which you can create PDF files with Arbortext Editor and Arbortext Styler or Arbortext Publishing Engine:

- With the APP publishing engine, you can publish PDF directly from an XML file or generate from intermediate PostScript source.
- With the FOSI publishing engine, you can publish PDF directly from an XML file.

Note

The FOSI and XSL-FO print engines are on sustained support and do not receive enhancements or maintenance fixes. APP is the recommended engine for print output.

See Choosing PDF Configuration Options on page 64 for information about how to set configuration choices.

See *Publishing Engine Overview* and *Print and PDF Configuration Files* in Arbortext Editor help for an explanation of print engines and PDF configuration files.

Using APP Publishing Engine for PDF

You may publish PDF or PostScript output. Separate configuration files support these actions.

For information about PDF configuration files supported by the APP print engine, please refer to PDF Configuration Files for APP on page 64

Using FOSI Publishing Engine for PDF

You can publish PDF with the FOSI publishing engine by sending publishing requests to Arbortext Publishing Engine and specifying the FOSI print engine in the stylesheet, by having Arbortext Styler running with Arbortext Editor, or by having a Desktop Composer license on Arbortext Editor.

Note

The FOSI and XSL-FO print engines are on sustained support and do not receive enhancements or maintenance fixes. APP is the recommended engine for print output.

Watermarks

Watermarks can underlay output pages for formatting, printing, or publishing a PDF. The methods for specifying watermarks for PDF differ between the Advanced Print Publisher and FOSI engines.

Watermarks when Publishing with APP

In Arbortext Styler, you may create a page region that holds text or a graphic, and set it to underlay the main content region of a page. Refer to *Defining Page Regions* in Arbortext Styler help for information on creating regions for a page set.

Watermarks when Publishing with FOSI

Set the *APTWATERMARKTEXT* environment variable to the value you want to appear as the watermark text. Refer to the *APTWATERMARKTEXT* online help topic for information on using it.

Creating PDF Bookmarks Using Arbortext Styler

PDF bookmarks created in this way are supported in PDF output generated by the APP, FOSI, and XSL-FO publishing engines.

In Arbortext Styler, you can elect to generate a table of contents (TOC) whose entries will form the bookmarks in your PDF. This can be separate from the main table of contents for the document if required and be configured to include your own set of entries. Refer to *Table of Contents Overview* in Arbortext Styler help for information.

Creating PDF Bookmarks Using FOSI

You can create bookmarks using markup illustrated in the following example in your stylesheet.

You can open the PDF to the first page, open the bookmark panel, and scale the page to fit in the window by placing the following anywhere in a document (or in a FOSI that places it anywhere in the document) using the following `atidmd:DocView` example:

```
<atidmd:DocumentMetaData source="atend">
 <atidmd:DocView bookmarks="auto" mode="bookmarks"
 fit="fitPage" destination="">
 </atidmd:DocView>
</atidmd:DocumentMetaData>
```

The destination attribute defaults to the page on which the atidmd:DocView tag appears. If a valid named destination name is placed in the destination attribute, the document will open at the page on which the named destination appears. A named destination can be created by inserting a link target at the desired point in the document, or adding an ID to a tag at that location.

Note

If DocView is specified in the stylesheet, it takes priority over the destination attribute value.

You can also generate an atidmd:DocumentMetaData node at the beginning of a document with a source="atend" attribute, and also an atidmd: DocumentMetaData node at the end of the root node content. Have the FOSI produce the following at beginning of the document, which has the effect of disabling automatic bookmarks:

```
<atidmd:DocumentMetaData source="atend">
 <atidmd:DocView mode="bookmarks" fit="fitPage" destination="">
 </atidmd:DocView></para>
</atidmd:DocumentMetaData>
```

Then have the FOSI produce the following at the end of document:

```
<atidmd:DocumentMetaData>
 <atidmd:Outline>
 <atidmd:Bookmark>
 <atidmd:Title>Book title</atidmd:Title>
 <atidmd:Bookmark>
 <atidmd:Title>Chapter 1 Title</atidmd:Title>
 </atidmd:Bookmark>
 <atidmd:Bookmark>
 <atidmd:Title>Chapter 2 Title</atidmd:Title>
 </atidmd:Bookmark>
 </atidmd:Bookmark>
 <atidmd:Bookmark state="closed">
 <atidmd:Title>List of Figures</atidmd:Title>
 <atidmd:Bookmark>
 <atidmd:Title>Figure 1</atidmd:Title>
 </atidmd:Bookmark>
 <atidmd:Bookmark>
 <atidmd:Title>Figure 2</atidmd:Title>
 </atidmd:Bookmark>
 </atidmd:Bookmark>
 </atidmd:Outline>
 </atidmd:DocumentMetaData>
```

The text variables used for links (such as `bookmarks.txt` in the following example) must be declared hotlinks to provide the bookmark destination using the _gtlink PI.

```
<stringdecl textid="docinfo.txt">
<stringdecl textid="chapter-bookmarks.txt" hotlink="1">
<stringdecl textid="book-title.txt">
<stringdecl textid="bookmarks.txt" hotlink="1">
<stringdecl textid="figures.bookmark.txt" hotlink="1">
Book eic:
 <usetext source='!<atidmd:DocumentMetaData source="atend"></atidmd:DocumentMetaDa
 placemnt="before"></usetext>
<savetext textid="chapter-bookmarks.txt" placemnt="before" conrule="\\">
<savetext textid="bookmarks.txt" placemnt="after"
 conrule='!<atidmd:Bookmark><atidmd:Title>!,book-title.txt,!</atidmd:Title>!,
 chapter-bookmarks.txt,!</atidmd:Bookmark>!'>
<usetext source='!<atidmd:DocumentMetaData><atidmd:DocInfo>!,docinfo.txt,!</atidmd
 <atidmd:Outline>!,bookmarks.txt,
 !<atidmd:Bookmark state="closed"><atidmd:Title>List of Figures</atidmd:Title>!,fi
 !</atidmd:Bookmark></atidmd:Outline></atidmd:DocumentMetaData>!'
 placemnt="after"></usetext>
```

Creating Document Properties

With Arbortext Styler

For information on how to define standard or user-defined metadata and document properties for a PDF file in your Arbortext Styler stylesheet, refer to *Passing Metadata to PDF Output* in Arbortext Styler help.

Metadata definitions in the `.style` file are supported in all print and PDF outputs — APP, FOSI, and XSL-FO.

With FOSI

You can generate document properties for the PDF by placing FOSI information anywhere in a document (or in a FOSI that places it anywhere in the document). An example showing the use of `atidmd:DocInfo` is given below:

```
<atidmd:DocumentMetaData>
 <atidmd:DocInfo>
 <atidmd:Entry>
 <atidmd:Key>Title</atidmd:Key><atidmd:Value>Moby Dick</atidmd:Value>
 </atidmd:Entry>
 <atidmd:Entry>
 <atidmd:Key>Author</atidmd:Key><atidmd:Value>Herman Melville</atidmd:Value>
 </atidmd:Entry>
 </atidmd:DocInfo>
<atidmd:DocumentMetaData></para>
```

Choosing PDF Configuration Options

When Arbortext Editor and Arbortext Publishing Engine publish PDF files, they use an XML configuration file (a `.appcf` for APP, a `.pdfcf` file for FOSI) to specify PDF options.

When publishing PDF, you can choose a PDF configuration file by:

- choosing a PDF configuration file in the **Publish PDF File** dialog box.
- specifying a default PDF configuration file using the `set pdfconfigfile` command (documented in the *Arbortext Command Language Reference*).

You can use or modify one of the PDF configuration files distributed with Arbortext Editor and Arbortext Publishing Engine or create a custom PDF configuration file.

Note

The FOSI and XSL-FO print engines are on sustained support and do not receive enhancements or maintenance fixes. APP is the recommended engine for print output.

PDF Configuration Files for APP

The distributed configuration files are:

- *Arbortext-path*`\app\standard.appcf`

  Supports publishing of PDF

- *Arbortext-path*`\app\postscript.appcf`

  Supports printing to a PostScript printer and generating a PostScript file

You can save a custom version of files if you wish to tailor your own PDF publishing process/output. Open the file in Arbortext Editor (without a stylesheet), make changes, then save the custom file, with the same file extension, in any of these locations:

- Publishing PDF from Arbortext Editor or Arbortext Styler — you can browse for a custom file, or locate it in the *APTCUSTOM*`\app` directory where APP will find it
- Publishing PDF via Arbortext Publishing Engine — a custom file must be located on the PE server, in any of these locations:
  - *Arbortext-path*`\app`
  - any *APTCUSTOM*`\app` directory

- an application or custom `doctype\nnn` directory, where *nnn* is the short doctype name of the doctype of the document being published.

Custom `.appcf` files must contain a single `Print` and a single `Format` element, although these do not require child elements to be valid.

See *PDF Configuration File for the APP Engine (.appcf)* in the *User's Guide to Arbortext Styler* and *Print and PDF Configuration Files* in Arbortext Editor help for information about custom configuration files.

PDF Configuration Files for FOSI

The distributed configuration files are:

- *Arbortext-path*`\lib\standard.pdfcf`

  This is the default PDF configuration file and configures the PDF file for general use. Embedding is turned on but also specifies a `NeverEmbed` list of the core 14 fonts.

  Compression is set to AUTO (choose the smaller of JPEG or ZIP compression).

  The target raster image resolution is 600 DPI for raster images exceeding the threshold of 900 DPI.

- *Arbortext-path*`\lib\screen.pdfcf`

  Configures the PDF file for screen display, with embedding turned on but also specifies a `NeverEmbed` list of the core 14 fonts.

  Compression is set to JPEG.

  The target raster image resolution is 300 DPI for raster images exceeding the threshold of 450 DPI.

- *Arbortext-path*`\lib\print.pdfcf`

  Optimizes PDF file for printing, with embedding turned on but also a `NeverEmbed` list of the core 14 fonts.

  Compression is set to ZIP (the default).

  The target raster image resolution is 1200 DPI for raster images exceeding the threshold of 1800 DPI.

- *Arbortext-path*`\lib\smallfile.pdfcf`

  Turns off embedding.

  Compression is set to JPEG.

  The target raster image resolution is 200 DPI for raster images exceeding the threshold of 300 DPI.

Note

PDF configuration files are distributed by locale and are located in *Arbortext-path*\lib\locale*lang*, where *lang* is the locale name.

You can create a custom PDF configuration file (*custom-file-name*.pdfcf) and put it in *Arbortext-path*\custom\lib, where it will be automatically accessible when Arbortext Editor or Arbortext Publishing Engine starts. You can also put a `set pdfconfigfile` statement in a custom ACL file placed in *Arbortext-path*\custom\init*custom-file-name*.acl, where it will be loaded at start time.

The structure and content of the `.pdfcf` PDF configuration file is explained in PDF DTD Element Usage (FOSI) on page 71 and the sections that follow it.

Linking Between PDF Files

The procedures that follow show how to link from a PDF to a page number and a named destination in another PDF file.

Note

Linking to a page number or named destination only applies when publishing to PDF directly from XML using Arbortext Publishing Engine.

To Link to a Page Number in the Target PDF

1. In Arbortext Editor, choose **Insert** ▸ **Link/Xref** and choose **Browse the Web** in the Resource Manager.If you are working in a non-DITA document, the menu option will be **Insert** ▸ **Link**, followed by **Web**.
2. Insert a URL that specifies the PDF file (the URL must include `.pdf`) followed by `#page=` and the page number (for example, `mydoc.pdf#page=30`).
3. Click the **Insert** button.

To Link to a Named Destination in the Target PDF

If the target is on an element that is also included in the table of contents, such as a `Title` tag, the link will target the table of contents instead of the location in the document. The two anchors have the same name, and only the first is recognized.

1. In the target document, select the element that will be the target of the link and give it an ID attribute, for example `mytarget`.If you are working in a non-DITA document, you can also use **Insert ▸ Link Target** to insert the ID.
2. In the source document, choose **Insert ▸ Link/Xref** and choose **Browse the Web** in the Resource Manager. If you are working in a non-DITA document, the menu option will be **Insert ▸ Link**, followed by **Web**.
3. Insert a URL that specifies the PDF file (the URL must include `.pdf`) followed by `#nameddest=` and the link target name (for example, `mydoc.pdf#nameddest=mytarget`).
4. Click the **Insert** button.

Configuring Security Options

If you want to apply security options to PDF files you create, you need to modify the PDF configuration file to specify the security options.

Security Options for APP Publishing

1. Make a copy of a PDF configuration file (you can choose `standard.appcf` from *`Arbortext-path`*`\app`).
2. Save the copy of the file to *`APTCUSTOM`*`\app`.
3. The `Security` element in the `.appcf` file includes options for configuring document protection. Add this element hierarchy to the required `Print` element: `Print PDFPrinter Security`
4. Use the attributes of the `Security` element to set security for your output PDF. For example, set the value of the *userPassword* or *masterPassword* attributes to the required string to provide password protection. Note that the attributes of the `Security` element are applied only if *masterPassword* is also set.
5. Save the file.
6. When publishing your PDF with the APP engine, select your custom configuration file in the **Config File** field of the **Publish to PDF File** dialog box.

Security Options for FOSI Publishing

PDF security options are explained in Security Element on page 82.

1. Make a copy of a PDF configuration file (you can choose a `.pdfcf` from *`Arbortext-path`*`\lib`).
2. Open Arbortext Architect, and choose **Edit ▸ PDFCF**. Open your `.pdfcf` file.
3. Locate the `Security` element, and choose **Edit ▸ Modify Attributes**.
4. Type a password as the **userPassword**. This restricts access to the PDF file to users who have the **userPassword**.
5. Type another password as the **masterPassword** if you plan to set any of the other security settings in the **Modify Attributes** dialog box to **yes**. For example, you can set **noPrint** to **yes** to prevent users from printing the PDF file or set **noModify** to **yes** to prevent users from modifying the PDF file.
6. Click **OK** to exit the **Modify Attributes** dialog box.
7. Choose **File ▸ Save**, and then **File ▸ Close** to exit Arbortext Architect.
8. Place the `.pdfcf` file in the `custom\lib` directory in the install tree.
9. Start Arbortext Editor and open a document.
10. Choose **File ▸ Publish ▸ PDF File**.
11. If you put your `.pdfcf` file in the `custom\lib`, you can select it from the Configuration File list .
12. Leave **View PDF File** selected, and then click **OK**.
13. When Adobe Acrobat opens the document, a prompt indicates that the file is protected and asks for a password. Type the password you set as the **userPassword** into the **Password** field.When the document opens, you can check that you are unable to perform the actions you prevented in the `.pdfcf` file. For example, if you set **noModify** to **yes**, the menu options for **Cut**, **Copy**, and **Delete** are unavailable.

Adding Fonts Used by Graphics

In `.eps`, `.drw`, and `.cgm` graphic formats, it's possible that a font within these files may not appear in PDF output, even though the graphic file has a properly defined font. There are three approaches for handling fonts within graphics:

- Confine fonts used in these graphics to the basic PDF 14 (Helvetica, Courier, Times-Roman, Symbol, and their variants), which are handled correctly.
- For fonts other than the basic PDF 14, embed the fonts in your `.eps`, `.drw`, and `.cgm` file . Embedded fonts are also handled correctly.
- For fonts other than the basic 14 that can't be embedded, use the procedure for font configuration that follows.

  If working with the APP engine, make the required changes in *`Arbortext-path`*`\pstill`.

  If working with the FOSI engine, you can copy the *`Arbortext-path`*`\pstill` directory to another location before making changes. If you do, set the *APTPSTILLPATH* environment variable to the path of the alternate location.

To Add PFA or PFB Fonts

1. In a command prompt window, navigate to the *`Arbortext-path`*`\pstill` directory.
2. Run the batch file `SetPath.bat` in the command window, from this directory. This file extracts the path to this `\pstill` directory.
3. If you have copied the `\pstill` directory elsewhere, navigate to that location.
4. Run `instfonts.exe` from the same command window as you ran `SetPath.bat`.
5. To add fonts, place PFA or PFB files in the `PSFonts` subdirectory. Then update the font files by entering:
   ```
   instfonts UPDATE
   ```
   The script updates the special font files used by PStill.

You can install a TTF file if it's not protected against conversion. Check your font license to see if it is allowed.

To Add TTF Fonts:

1. In a command prompt window, navigate to the *`Arbortext-path`*`\pstill` directory.
2. Run the batch file `SetPath.bat` in the command window, from this directory. This file extracts the path to this `\pstill` directory.
3. If you have copied the `\pstill` directory elsewhere, navigate to that location.

4. Run `instfonts.exe` from the same command window as you ran `SetPath.bat`.
5. Install a TTF font by entering:
   `instfonts TTFFULLINSTALL` *`full-path-to-TTF-file postscript-name`*
   or, you can enter:
   `instfonts TTFINSTALL` *`full-path-to-TTF-file postscript-name`*
   `instfonts UPDATE`

   The PostScript name is the name referenced in the `.eps`, `.drw`, or `.cgm` file. The name is case-sensitive.

To Create Fonts for the Basic 14 PDF Fonts:

1. In a command prompt window, navigate to the *`Arbortext-path`*`\pstill` directory.
2. Run the batch file `SetPath.bat` in the command window, from this directory. This file extracts the path to this `\pstill` directory.
3. If you have copied the `\pstill` directory elsewhere, navigate to that location.
4. Run `instfonts.exe` from the same command window as you ran `SetPath.bat`.
5. Create font files for the basic PDF 14 fonts by entering:
   `instfonts CREATE`

Configuring Fonts for FOSI Publishing

The `Font` element in the PDF configuration file lets you configure font locations, map `.tfm` file names to fonts, and enable font substitutions, embedding, and subsets. Consult the PDF DTD Element Usage (FOSI) on page 71 for detailed information on the elements and their attributes.

Arbortext Command Language and the Arbortext Publishing Engine use font metrics to lay out the text of a document into paragraphs and pages. However, a real font is necessary for rendering the page. A real font is a raster or vector font. A raster font (a `.pk` file) contains pixels for all characters in the font rasterized to a particular resolution. A vector font (a type 1 or true type font) contains instructions for drawing the character outlines.

The `FontName` element specifies real fonts. These font names may contain Unicode characters. The `FontName` element has a `select` attribute that sets the data condition for a particular operating system. If you use `select`, the value must match one of the tokens generated by the application (for example `Windows`), called the selection criteria. A `select` attribute may include more than one token. If any of them is equal to any of the selection criteria tokens, then

the FontName matches. If a select attribute is empty or not present, then a match is assumed. The encoding attribute specifies the encoding to use with the specified font. Eight-bit and multi-byte formats are supported.

The FontName element can name a system font or a font outline file (TTF, PFA, or PFB). The type of file is specified in the type attribute, and the path and file name is specified in the path attribute. If the type is SYS, the path is ignored. If path contains a relative path, the search is relative to the custom\fonts subdirectory. The contents of the FontName tag for a font outline file should be the name of the font as represented in the outline file.

The FontName element has a simulate attribute that can specify the kind of simulation (such as bold or italic) to apply to the font. The simulateMode attribute specifies how to apply the simulation, either by modifying the font display within the PDF (by displaying characters at an angle for italic or through multiple registrations for bold) or by setting a flag that tells the PDF viewer to render the simulation.

The Map element associates the name of a .tfm file with a font. Multiple real fonts may be listed, but Arbortext Editor and Arbortext Publishing Engine use only the first match. This allows the select attribute to determine which font is used on a given platform.

For fonts that are not specified in FontName, are not mapped in this file, and do not have the specified font face available, you can use the Simulation elements Bold and Italics to simulate bold and italic faces. Use simulated bold, italic, and bold-italic font styles for CJK fonts and Arial Unicode MS, which don't have bold or italic faces.

The Locations element can specify directories where the direct PDF process searches for font files.

The FontPath element specifies PK raster font files. The dpi attribute specifies the resolution of the fonts. The contents of the element are the path to the .pk file relative to the pixels directory.

You can specify Substitute to use a different font in the PDF file than was used in the original file. The substitute font must have the same font metrics as the original font. Arbortext Editor and Arbortext Publishing Engine use the first replacement that matches the selection criteria.

Embedding lets you select the fonts that are embedded or prevent embedding in the PDF file. The subsetting attribute allows you to control whether to embed the whole font or just the characters needed by the PDF file.

PDF DTD Element Usage (FOSI)

FOSI publishing supports the elements described in the following sections for configuring PDF options. The document type for FOSI PDF configuration files (.pdfcf) is located at:

Arbortext-path\doctypes\pdfconfig\pdfconfig.dtd

The elements are organized according to the hierarchy within the DTD.

Pdfconfig Document Type

The <Pdfconfig> document type is a PDF configuration XML document.

The <Pdfconfig> document type can have the following child elements:

Child elements of Pdfconfig

General Element on page 72	Optional
Color Element on page 83	Optional
Font Element on page 86	Optional
Documentation Element on page 98	Optional
Label Element on page 98	Optional

General Element

The <General> element controls a variety of aspects of the PDF file.

The <General> element has the following child elements:

Child elements of General

Annotations Element on page 73	Optional and may be used once
Compatibility Element on page 73	Optional and may be used once
Compression Element on page 75	Optional and may be used once
Cropmarks Element on page 75	Optional and may be used once
Docinfo Element on page 77	Optional and may be used once
Images Element on page 78	Optional and may be used once

Child elements of General (continued)

Merge Element on page 78	Optional and may be used once
Open Element on page 79	Optional and may be used once
Security Element on page 82	Optional and may be used once

The <General> element has one attribute, fixupPageSizes = yes | no. If set to yes, it reconciles differences in page size between the front and back of a sheet. The height will be the larger of the two page heights and the width will be the larger of the two page widths. The default is no.

If a page dimension is increased, the content of the page will be centered within that dimension. Page dimensions are never decreased.

Annotations Element

The <Annotations> element controls the display f bookmarks, links, and other features in the PDF file. It's optional and may be used once.

The <Annotations> element has no child elements.

The <Annotations> element has the following attributes:

Attributes of Annotations

Attribute and values	Description
enabled = yes | no	Allows you to turn off PDF features that aren't relevant for printing, such as bookmarks and links. The default is yes.
nameddestToPage = yes | no	Setting to yes converts named destinations to page numbers in PDF links.

Compatibility Element

The <Compatibility> element specifies the type of PDF file that is produced. It's optional and may be used once.

The <Compatibility> element may have either a <PDF> or <PDFX> child element.

The <Compatibility> element has no attributes.

PDF Element

The <PDF> element specifies the version of the PDF file.

The <PDF> element has no child elements.

The <PDF> element has one attribute, *level* = 1.3 | 1.4 | 1.5 | 1.6. The default is 1.4. PDF versions 1.3, 1.4, 1.5, 1.6, and 1.7 are supported by Adobe Acrobat 4.0, 5.0, 6.0, 7.0, and 8.0 respectively.

See PDF Compatibility / Version for the PDF compatibility details.

PDFX Element

The <PDFX> element specifies the PDF/X standards series, which provides a consistent and robust subset of PDF which can be used to deliver data suitable for commercial printing. The driver can generate output conforming to the following variations of PDF/X:

- PDF/X-1 and PDF/X-1a, both defined in ISO 15930-1:2001
- PDF/X-3 as defined in ISO 15930-3:2002

PDF/X is specified using the <PDFX> tag within the <Compatibility> tag. If you specify <PDFX>, you need to set the enabled attribute of the <Annotations> element to no.

Note

PDF/X support is offered as a technology preview. No guarantees are made as to the correctness or usability of output using the PDF/X options.

For more information on using PDF/X, refer to PDF/X Frequently Asked Questions available on Adobe website.

The <PDFX> element has no child elements.

Attributes of PDFX

Attribute and values	Description
level = 1:2001 | 1a:2001 | 1a:2003 | 2:2003 | 3:2002 | 3:2003	settings beginning with 1 specify PDF/X-1, settings beginning with 1a specify PDF/X-1a, and settings beginning with 3 specify PDF/X-3. The default is 3:2003.
outputIntent = *CDATA*	Specifies the rendering intent.
defaultRGB = *CDATA*	Specifies the ICC profile for converting RGB images, text, and graphics.

Attributes of PDFX (continued)

Attribute and values	Description
defaultGray = CDATA	Specifies the ICC profile for converting Gray images, text, and graphics.
defaultCMYK = CDATA	Specifies the ICC profile for converting CMYK images, text, and graphics.

Compression Element

The `<Compression>` element specifies the type and level of compression. It's optional and may be used once.

The `<Compression>` element has no child elements.

The `<Compression>` element has the following attributes:

Attributes of Compression

Attribute and values	Description
level = NMTOKEN	Specifies the ZIP compression level, which you can set from 0 (none) to 9 (maximum). The default is `6`. This attribute is ignored if *type* = `JPEG`.
quality = NMTOKEN	Specifies the JPEG quality, which you can set from 1 (lowest) to 100 (highest). The default is `80`. This attribute is ignored if *type* = `ZIP`.
type = `ZIP` | `JPEG` | `AUTO`	Specifies the kind of compression. `AUTO` chooses whichever is smaller between `ZIP` and `JPEG`. The default is `ZIP`.

Cropmarks Element

The <Cropmarks> element specifies the characteristics of crop marks and whether they appear in the PDF output. It's optional and may be used once.

The <Cropmarks> element has no child elements.

The <Cropmarks> element has the following attributes:

Attributes of Cropmarks

Attribute and values	Description
enabled = `yes` | `no`	Specifies whether to display crop marks in the output. The default is `yes`.
pageDims = `absolute` | `increment`	When set to `absolute`, uses the values set by *pageWidth* and *pageHeight* for the output page dimension. When set to `increment`, *pageWidth* and *pageHeight* are added to the input page dimension to get the output page dimension. The default is `increment`.
pageWidth = *CDATA*	Specify an integer as the output page width in points; used as input for *pageDims* to determine the page dimension. The default is `144`, which is the equivalent of 2 inches.
pageHeight = *CDATA*	Specify an integer as the output page height in points; used as input for *pageDims* to determine the page dimension. The default is `144`, which is the equivalent of 2 inches.
gap = *CDATA*	Specify an integer as the distance in points from the corners of content to each crop mark. The default is `4`.
thickness = *CDATA*	Specify an integer as the rule thickness in points for the crop mark. Decimal values are allowed. The default is `.25`.
length = *CDATA*	Specify an integer as the size in points for the length of the crop mark. The default is `36`.
placement = `center` | `upperLeft` | `upperRight` | `lowerLeft` | `lowerRight` | `useOffsets`	Specifies where the page content is placed. Specifying `useOffsets` positions the upper left corner of the contents `xOffset` points down from top of page and `yOffset` points from the left. The default is `center`, which centers the content on the page.

Attributes of Cropmarks (continued)

Attribute and values	Description
xOffset = *CDATA*	Specifies the vertical distance in points for placing the upper left corner of content. This value is ignored unless *placement* = `useOffsets`.
yOffset = *CDATA*	Specifies the horizontal distance in points for placing the upper left corner of content. This value is ignored unless *placement* = `useOffsets`.

Docinfo Element

The `<Docinfo>` element specifies document properties in the PDF being created. It's optional and may be used once.

The `<Docinfo>` element has one required child element, `<Entry>`, which specifies the document property names and their values.

The `<Docinfo>` element has no attributes.

Entry Element

The `<Entry>` element specifies the document properties that can be set when the PDF is created. `<Entry>` is required and repeatable.

The `<Entry>` element has no child elements.

The `<Entry>` element has the following attributes:

Attributes of the Entry element

Attributes and values	Description
key = *CDATA*	Specifies the document property name. A *key* is case-sensitive (i.e. Title, not title). Key names and values are associated with the Title, Author, Subject, and Keywords fields of Document Properties. Other names and values will be displayed on the Custom tab of Document Properties for the PDF.
value = *CDATA*	Specifies the document property value.

Images Element

The <Images> element specifies how graphics are handled in the PDF file. It's optional and may be used once.

The <Images> element has one optional child element, <DownSample>.

The <Images> element has two attributes:

Attributes of the Images element

Attribute and values	Description
passthrough = bmp | BMP | gif | GIF | jpg | JPG | jpeg | JPEG | png | PNG | tif | TIF | tiff | TIFF	specifies the graphic types to be passed through to the PDF document without processing (as long as cropping is not required).
rasterize = cgm | CGM	specifies the graphic type to be rasterized. Currently only supports CGM graphics.

DownSample Element

The <DownSample> element controls how raster images are handled. It's optional and may be used once.

The <DownSample> element has no child elements.

The <DownSample> element has the following attributes and values:

Attributes of the DownSample element

Attribute and values	Description
targetDpi = *NMTOKEN*	Specifies the dots per inch (DPI) that a graphic's resolution will be reduced to when its resolution is larger than the value specified by the threshold attribute.
threshold = *NMTOKEN*	When a graphic's resolution is larger than the specified value, the resolution of the graphic is reduced to the targetDpi value.

Merge Element

The <Merge> element allows inserting existing PDFs into the PDF being created. It's optional and may be used once.

The <Merge> element has one required child element, <Insert>, which specifies the insertion instructions.

The <Merge> element has no attributes.

Insert Element

The <Insert> element specifies the instructions for inserting existing PDFs into the PDF being created. By default, PDFs are inserted on a recto page with even padding at the end of the document. If no destination or placement is provided, then the PDF will be inserted after the entire document that is being published to PDF. <Insert> is required and repeatable.

The <Insert> element has no child elements.

The <Insert> element has the following attributes:

Attributes of the Insert element

Attribute and values	Description
path = *CDATA*	Specifies the path and file name of the PDF document to insert.
start = `recto` | `verso` | `none`	Specifies the page layout position for inserting the PDF. The default is `recto`.
pad = `even` | `odd` | `none`	Specifies the padding to use to complete the inserted section.
destination = *CDATA*	Specifies the target destination within the PDF where the referenced PDF should be inserted.
placement = `before` | `after`	Specifies whether to place an inserted PDF `before` or `after` the entire document.

Open Element

The <Open> element controls the characteristics of the PDF file when it is opened. It's optional and may be used once.

The <Open> element has no child elements.

The <Open> element has the following attributes:

Attributes of the Open element

Attribute and values	Description
mode = `none` | `bookmarks` | `thumbnails` | `fullscreen`	Specifies a method for displaying the document, including settings for the navigation pane. The default is `none`.
fit = `fitPage` | `fitWidth` | `actualSize`	Specifies the initial magnification view in the PDF viewer relative to its display area. *fit* is ignored if *mode* is `fullscreen`. `fitPage` displays the entire page in the window. `fitWidth` displays the page scaled to the width of the window. `actualSize` displays the page at 100%. The default is `fitPage`.
destination = *CDATA*	Specifies the page displayed when the document is opened. Specify a named destination or a page number. A page number must be preceded by `page=`. The default is the first page.
displayTitle = `yes` | `no`	By default, Adobe Acrobat displays the PDF file name in the title bar. If *displayTitle* is set to `yes`, Acrobat will display the document title instead.
pageLayout = `default` | `single` | `twoup`	Setting *pageLayout* to `default` displays the document according to Acrobat's settings for opening a document. Setting *pageLayout* to `single` displays a single page when the document is opened. Setting *pageLayout* to `twoup` displays two pages side-by-side when the document is opened. The default is `single`.

Attributes of the Open element (continued)

Attribute and values	Description
continuous = `yes` | `no`	The setting of *continuous* is ignored if *pageLayout* is set to default. Otherwise, if *continuous* is set to `no`, only a single page or pair of pages will be shown at any given time. Scrolling to the bottom will cause an abrupt change to the next page or pages. If set to `yes`, the page transitions will be shown. For PDF versions 1.4 or lower, *continuous*=`no` is not supported for *pageLayout*=`twoup`.
facing = `yes` | `no`	The setting of *facing* is ignored unless *pageLayout* is set to `twoup`. Otherwise, if *facing* is `no`, the document will be displayed starting with the first page on the left. Each pair of pages will be the recto and verso of a sheet. If *facing* is `yes`, the first page will appear by itself and subsequent pairs of pages will be the verso of one page and the recto of the next page.

Use the following guidelines:

- Setting *pageLayout*=`default` ignores the other settings.
- For a single page non-scrolling view, use *pageLayout*=`single` and *continuous*=`no` (ignores *facing*).
- For a single page scrolling view, use *pageLayout*=`single` and *continuous* =`yes` (ignores *facing*).
- For a two page scrolling view that starts with first page on the left side, use *pageLayout*=`twoup`, *continuous*=`yes`, and *facing*=`no`.
- For a two page scrolling view that starts with first page on the right side and then continues with facing pages, use *pageLayout*=`twoup`, *continuous*=`yes`, and *facing*=`yes`.
- For a two page non-scrolling view that starts with the first page on the left side, use *pageLayout*=`twoup`, *continuous*=`no`, and *facing*=`no`.
- For a two page non-scrolling view that starts with the first page on the right side and then continues with facing pages, use *pageLayout*=`twoup`, *continuous*=`no`, and *facing*=`yes`.

Security Element

The <Security> element limits access to the PDF file. It's optional and may be used once.

The <Security> element has no child elements.

The <Security> element has the following attributes and values:
Note that these attributes are applied only if *masterPassword* is also set.

Attributes of the Security element

Attribute and values	Description
userPassword = *CDATA*	Specifies a user password that is needed to view the PDF file. **Note** When using a password, the PDF file is encoded using 128-bit encryption (40-bit encryption when 1.3 compatibility is used).
masterPassword = *CDATA*	Specifies a password to override security restrictions (*noPrint*, *noModify*, *noCopy*, *noAnnots*, *noForms*, *noAccessible*, *noAssemble*, *noHiresPrint*) that are applied when the PDF file is created This password must be different than the user password.
noPrint = yes | no	When set to yes, prevents printing of the PDF file. You must also specify a *masterPassword* when this attribute is set to yes.
noModify = yes | no	When set to yes, prevents modifying of the PDF file. You must also specify a *masterPassword* when this attribute is set to yes.
noCopy = yes | no	When set to yes, prevents copying and extracting text and graphics, and disables the accessibility interface of the PDF file. You must also specify a *masterPassword* when this attribute is set to yes.
noAnnots = yes | no	When set to yes, prevents adding or changing comments or form fields in the

Attributes of the Security element (continued)

Attribute and values	Description
	PDF File. You must also specify a *masterPassword* when this attribute is set to `yes`.
noForms = `yes` | `no`	When set to `yes`, prevents changing form fields in the PDF file. You must also specify a *masterPassword* when this attribute is set to `yes`.
noAccessible = `yes` | `no`	When set to `yes`, prevents extracting text and graphics in the PDF file for accessibility purposes (such as for a screen reader program). You must also specify a *masterPassword* when this attribute is set to `yes`.
noAssemble = `yes` | `no`	When set to `yes`, prevents inserting, deleting, or rotating pages, and creating bookmarks and thumbnails in the PDF file. You must also specify a *masterPassword* when this attribute is set to `yes`.
noHiresprint = `yes` | `no`	When set to `yes`, prevents high-resolution printing of the PDF file. If *noprint* = `yes`, printing is restricted to the "print as image" option. You must also specify a *masterPassword* when this attribute is set to `yes`.

For detailed information about attribute inter-dependencies see Adobe documentation.

Color Element

The `<Color>` element controls how color is handled on a black and white printer. It's optional and may be used once.

The `<Color>` element has one optional child Convert Element on page 84.

The `<Color>` element has the following attributes:

Attributes of the Color Element

Attribute and values	Description
monochrome = `yes` | `no`	Setting to `yes` replaces all foreground colors (except white) with black, and background colors are rendered as shades of gray.
defaultRGB = *CDATA*	Specifies the ICC profile for converting RGB images, text, and graphics.
defaultGray = *CDATA*	Specifies the ICC profile for converting Gray images, text, and graphics.
defaultCMYK = *CDATA*	Specifies the ICC profile for converting CMYK images, text, and graphics.

The default input from Arbortext Editor is RGB, so all text is RGB. Graphics are not converted.

The color profile settings (*defaultRGB*, *defaultGray*, *defaultCMYK*) for the `Color` element override the color profile settings for the `<PDFX>` element. You can set *defaultRGB* = `sRGB` to revert to the color handling used in versions prior to 5.2 of Arbortext Editor and Arbortext Publishing Engine.

Convert Element

The `<Convert>` element specifies the mapping of color conversion. It's optional and may be used once.

The `<Convert>` element may have the optional `<Model>` and `<Spot>` child elements. The `<Hue>` element is not implemented.

The `<Convert>` element has no attributes.

Model Element

The `<Model>` element transforms any color from the source color model to the target color model. It's optional and repeatable.

The `<Model>` element has no child elements.

The `<Model>` element has the following attributes:

Attributes of the Model element

Attribute and values	Description
from = `RGB` | `CMYK` | `Grayscale` | `all`	The only source color model supported is `RGB`. Specifying *from* = `all` only converts RGB source.
to = `RGB` | `CMYK` | `Grayscale`	The target model supports only `Grayscale` or `CMYK`.

Spot Element

The `<Spot>` element specifies the mapping of one specified color to another specified color. It's optional and repeatable.

The `<Spot>` element specifies the mapping using the child elements `<CMYK>`, `<Grayscale>`, or `<RGB>`. Specify the mapping by using the form *source,target*. The *source* color will be replaced with the *target* color.

The `<Spot>` element has no attributes.

CMYK Element

The `<CMYK>` element specifies the eight digit hexadecimal representation of a color based on eight bits each of cyan, magenta, yellow, and black.

The `<CMYK>` element has no child elements.

The `<CMYK>` element has no attributes.

Grayscale Element

The `<Grayscale>` element specifies the three digit hexadecimal representation of grayscale based on twelve bit grayscale value.

The `<Grayscale>` element has no child elements.

The `<Grayscale>` element has no attributes.

RGB Element

The `<RGB>` element specifies the six digit hexadecimal representation of a color based on eight bits each of red, green, and blue.

The `<RGB>` element has no child elements.

The `<RGB>` element has no attributes.

Font Element

The <Font> element configures font locations, maps .tfm file names to fonts, and enables font substitutions, embedding and subsets.

The <Font> element has the following child elements:

Child elements of the Font element

DefaultFont Element on page 86	Optional and may be used once
EmbedAlways Element on page 88	Optional and may be used once
EmbedNever Element on page 90	Optional and may be used once
Locations Element on page 92	Optional and may be used once
Map Element on page 92	Optional and repeatable
Simulation Element on page 94	Optional and may be used once
Substitute Element on page 96	Optional and repeatable

The <Font> element has one attribute, bitmapResolution. The default target resolution for bitmap fonts is 600 DPI.

DefaultFont Element

The <DefaultFont> element specifies the default font used when creating PDF files. It's optional and may be used once.

The <DefaultFont> element has one optional child element, <FontName>.

The <DefaultFont> element has no attributes.

FontName Element

The content of the <FontName> element specifies a system or font outline file. It's optional and repeatable.

The <FontName> element has no child elements. The <Simulation> element overrides <FontName>.

The <FontName> element has the following attributes:

Attributes of the FontName element

Attribute and values	Description
select = *CDATA*	Specifies the operating system on which the `FontName` is used. The value must match one of the tokens generated by the application (for example, “Windows”). You can specify more than one selection. If any of the selections is equal to any of the selection criteria tokens, then the `FontName` matches. If this attribute is empty or not present, a `FontName` match is assumed.
encoding = *CDATA*	Indicates the encoding to be used with the specified font. Multi-byte (Unicode and PostScript CMaps) and 8-bit (cp125x, where x = 0-8; iso8859-x, where x = 1-10, 13-16) formats are supported.
simulate = `normal` | `bold` | `italic` | `bolditalic`	Specifies a font face to simulate. For example, if a font does not have a bold face font, you could specify `bold`. The default is `normal`.
simulateMode = `PDF` | `reader`	Specifies how to apply the simulation. `PDF` modifies the font within the PDF, by slanting the output for italics or using multiple registrations of characters for bold. `reader` sets a flag that tells the PDF viewer to render the simulation, which will work with standard PDF fonts. The default is `reader`. **Note** If you choose `reader`, it only applies to standard PDF fonts, which have metrics built into the PDF reader.
type = `SYS` | `TTF` | `PFA` | `PFB`	Specifies the type of font, system (SYS) or a font outline file (`.ttf`, `.pfa`, or `.pfb` file). The default is SYS.

Attributes of the FontName element (continued)

Attribute and values	Description
path = *CDATA*	Specifies the path and file name to the type of file (`.ttf`, `.pfa`, or `.pfb`) specified by *type*. If *type* = `SYS`, the path is ignored. If you specify a relative path, the path is relative to the `custom` directory.
metrics = *CDATA*	Specifies the metrics file (`.afm` or `.pfm`) to be used when *type* = `PFA` or `PFB`. If you do not specify a metrics file, Arbortext Editor searches for a `.afm` file with the same base file name as the `PFA` in the specified *path*.

EmbedAlways Element

The `<EmbedAlways>` element specifies the fonts that are embedded in the PDF file. However, `<EmbedNever>` takes precedence over `<EmbedAlways>`.

The `<EmbedAlways>` element has one child element, `<FontName>`.

The `<EmbedAlways>` element has the following attributes and values:

Attributes of the EmbedAlways element

Attribute and values	Description
allFonts = `yes` | `no`	When set to `yes`, embeds all fonts except fonts specified in `<EmbedNever>`. If you want to embed some fonts, list them in the `FontName` child element.
subsetting = `yes` | `no`	When set to `yes`, embed only the characters needed in a particular PDF file, based on the percentage of the font used. When set to `no`, embed the whole font.

Attributes of the EmbedAlways element (continued)

Attribute and values	Description
subsetPercent = *NMTOKEN*	Controls whether the whole font is embedded (100%) or just the characters needed by the PDF file, specified by a percentage of the font used. If the number of characters used exceeds this number, the entire font is embedded. (See the `screen.pdfcf` file for an example.)
ifEmbedFails = `ignore` | `warn` | `abort`	Controls what happens in the PDF creation process when embedding fails.

FontName Element

The content of the `<FontName>` element specifies a system or font outline file. It's optional and repeatable.

The `<FontName>` element has no child elements. The `<Simulation>` element overrides `<FontName>`.

The `<FontName>` element has the following attributes:

Attributes of the FontName element

Attribute and values	Description
select = *CDATA*	Specifies the operating system on which the `FontName` is used. The value must match one of the tokens generated by the application (for example, "Windows"). You can specify more than one selection. If any of the selections is equal to any of the selection criteria tokens, then the `FontName` matches. If this attribute is empty or not present, a `FontName` match is assumed.
encoding = *CDATA*	Indicates the encoding to be used with the specified font. Multi-byte (Unicode and PostScript CMaps) and 8-bit (cp125x, where x = 0-8; iso8859-x, where x = 1-10, 13-16) formats are supported.
simulate = `normal` | `bold` | `italic` | `bolditalic`	Specifies a font face to simulate. For example, if a font does not have a bold face font, you could specify `bold`. The default is `normal`.

Attributes of the FontName element (continued)

Attribute and values	Description
simulateMode = `PDF` | `reader`	Specifies how to apply the simulation. `PDF` modifies the font within the PDF, by slanting the output for italics or using multiple registrations of characters for bold. `reader` sets a flag that tells the PDF viewer to render the simulation, which will work with standard PDF fonts. The default is `reader`. **Note** If you choose `reader`, it only applies to standard PDF fonts, which have metrics built into the PDF reader.
type = `SYS` | `TTF` | `PFA` | `PFB`	Specifies the type of font, system (SYS) or a font outline file (`.ttf`, `.pfa`, or `.pfb` file). The default is SYS.
path = *CDATA*	Specifies the path and file name to the type of file (`.ttf`, `.pfa`, or `.pfb`) specified by *type*. If *type* = `SYS`, the path is ignored. If you specify a relative path, the path is relative to the `custom` directory.
metrics = *CDATA*	Specifies the metrics file (`.afm` or `.pfm`) to be used when *type* = `PFA` or `PFB`. If you do not specify a metrics file, Arbortext Editor searches for a `.afm` file with the same base file name as the `PFA` in the specified *path*.

EmbedNever Element

The `<EmbedNever>` element specifies the fonts that you do not want embedded in the PDF file. When `<EmbedAlways>` is specified, `<EmbedNever>` takes precedence over `<EmbedAlways>`.

The `<EmbedNever>` element has one child element, `<FontName>`.

The `<EmbedNever>` element has no attributes.

FontName Element

The content of the <FontName> element specifies a system or font outline file. It's optional and repeatable.

The <FontName> element has no child elements. The <Simulation> element overrides <FontName>.

The <FontName> element has the following attributes:

Attributes of the FontName element

Attribute and values	Description
select = *CDATA*	Specifies the operating system on which the FontName is used. The value must match one of the tokens generated by the application (for example, "Windows"). You can specify more than one selection. If any of the selections is equal to any of the selection criteria tokens, then the FontName matches. If this attribute is empty or not present, a FontName match is assumed.
encoding = *CDATA*	Indicates the encoding to be used with the specified font. Multi-byte (Unicode and PostScript CMaps) and 8-bit (cp125x, where x = 0-8; iso8859-x, where x = 1-10, 13-16) formats are supported.
simulate = normal | bold | italic | bolditalic	Specifies a font face to simulate. For example, if a font does not have a bold face font, you could specify bold. The default is normal.
simulateMode = PDF | reader	Specifies how to apply the simulation. PDF modifies the font within the PDF, by slanting the output for italics or using multiple registrations of characters for bold. reader sets a flag that tells the PDF viewer to render the simulation, which will work with standard PDF fonts. The default is reader. **Note** If you choose reader, it only applies to standard PDF fonts, which have metrics built into the PDF reader.
type = SYS | TTF | PFA | PFB	Specifies the type of font, system (SYS) or

Attributes of the FontName element (continued)

Attribute and values	Description
	a font outline file (`.ttf`, `.pfa`, or `.pfb` file). The default is SYS.
path = *CDATA*	Specifies the path and file name to the type of file (`.ttf`, `.pfa`, or `.pfb`) specified by *type*. If *type* = `SYS`, the path is ignored. If you specify a relative path, the path is relative to the `custom` directory.
metrics = *CDATA*	Specifies the metrics file (`.afm` or `.pfm`) to be used when *type* = `PFA` or `PFB`. If you do not specify a metrics file, Arbortext Editor searches for a `.afm` file with the same base file name as the `PFA` in the specified *path*.

Locations Element

The `<Locations>` element can provide additional directories to add to the search path for locating font files such as AFM, PFA, TTF, and so on. It's optional and can only be used once.

The `<Locations>` element has one optional child element, `<Path>`.

The `<Locations>` element has no attributes.

Path Element

The `<Path>` element supplies a directory to add to the search path for locating font files such as AFM, PFA, TTF, and the like. It's optional and repeatable.

The `<Path>` element has no child elements.

The `<Path>` element has no attributes.

Map Element

The `<Map>` element maps the `.tfm` file names to fonts that will be embedded in the PDF file. It's optional and repeatable.

The `<Map>` element must specify one of the child elements, `<FontName>` or `<FontPath>`.

The `<Map>` element has the following attributes and values:

Attributes of the Map element

Attribute and values	Description
tfm = *CDATA*	Associates a `.tfm` file with a real font. Although you can list multiple real fonts, only the first match is used.
adj = `yes` | `no`	Not supported.

FontName Element

The content of the `<FontName>` element specifies a system or font outline file. It's optional and repeatable.

The `<FontName>` element has no child elements. The `<Simulation>` element overrides `<FontName>`.

The `<FontName>` element has the following attributes:

Attributes of the FontName element

Attribute and values	Description
select = *CDATA*	Specifies the operating system on which the `FontName` is used. The value must match one of the tokens generated by the application (for example, "Windows"). You can specify more than one selection. If any of the selections is equal to any of the selection criteria tokens, then the `FontName` matches. If this attribute is empty or not present, a `FontName` match is assumed.
encoding = *CDATA*	Indicates the encoding to be used with the specified font. Multi-byte (Unicode and PostScript CMaps) and 8-bit (cp125x, where x = 0-8; iso8859-x, where x = 1-10, 13-16) formats are supported.
simulate = `normal` | `bold` | `italic` | `bolditalic`	Specifies a font face to simulate. For example, if a font does not have a bold face font, you could specify `bold`. The default is `normal`.
simulateMode = `PDF` | `reader`	Specifies how to apply the simulation. `PDF` modifies the font within the PDF, by slanting the output for italics or using multiple registrations of characters for

Attributes of the FontName element (continued)

Attribute and values	Description
	bold. `reader` sets a flag that tells the PDF viewer to render the simulation, which will work with standard PDF fonts. The default is `reader`. **Note** If you choose `reader`, it only applies to standard PDF fonts, which have metrics built into the PDF reader.
type = `SYS` | `TTF` | `PFA` | `PFB`	Specifies the type of font, system (SYS) or a font outline file (`.ttf`, `.pfa`, or `.pfb` file). The default is SYS.
path = *CDATA*	Specifies the path and file name to the type of file (`.ttf`, `.pfa`, or `.pfb`) specified by *type*. If *type* = `SYS`, the path is ignored. If you specify a relative path, the path is relative to the `custom` directory.
metrics = *CDATA*	Specifies the metrics file (`.afm` or `.pfm`) to be used when *type* = `PFA` or `PFB`. If you do not specify a metrics file, Arbortext Editor searches for a `.afm` file with the same base file name as the `PFA` in the specified *path*.

FontPath Element

The `<FontPath>` element specifies PK fonts (`.pk` files). The contents of the `<FontPath>` element specify the path and file name of the `.pk` file relative to the `pixels` directory.

The `<FontPath>` element has no child elements.

The `<FontPath>` element has one attribute, *dpi* = *CDATA*. The *dpi* attribute specifies the font resolution.

Simulation Element

The `<Simulation>` elements can apply bold or italic simulation to fonts that are not specified in `FontName`, are not mapped in this configuration file, and do not have the specified font face available. It's optional and can only be used once.

The <Simulation> element has the optional child elements Bold and Italics.

The <Simulation> element has no attributes.

Bold Element

The <Bold> element controls simulation of fonts that do not have the specified font face available. Bold simulation is controlled by this element if a font was specified in the <FontName> element, with its simulateMode attribute set to PDF and its simulate attribute set to bold. It's optional and can only be used once.

If the font was not specified in a <FontName> element, the <Bold> element controls bold simulation if no bold face is found for the font.

The <Bold> element has no child elements.

The <Bold> element has the following attributes:

Attributes of the Bold element

Attribute and values	Description
enable = yes | no	Specifies whether to apply bold simulation. The default is yes.
percent = *CDATA*	Specifies the percentage of the font size to use as the offset for multiple registrations of the font for simulating bold. The default is 5 percent.
threshold = *CDATA*	Specifies the font point size above which the offset will be constant to avoid problems with large offsets on large font sizes. The threshold value will be used to calculate the offset. The default is 14 points.

Italics Element

The <Italics> element controls simulation of fonts that do not have the specified font face available. Italics simulation is controlled by this element if a font was specified in the <FontName> element, with its simulateMode attribute set to PDF and its simulate attribute set to italic. It's optional and can only be used once.

If the font was not specified in a <FontName> element, the <Italics> element controls italics simulation if no italic face is found for the font.

The <Italics> element has no child elements.

The <Italics> element has the following attributes:

Attributes of the Italics element

Attributes and values	Description
enable = = yes | no	Specifies whether to apply italics simulation. The default is yes.
angle = *CDATA*	Specifies the angle of the text to simulate italic. The setting is applied counterclockwise from a vertical position. The default value is -18.8 degrees.

Substitute Element

The <Substitute> element specifies a different font for producing the PDF file than was used in the original document. The replacement font must have the same font metrics as the original font. The original font name should be followed by the name of one or more real font names. The first matching replacement font found will be used. Arbortext Editor and Arbortext Publishing Engine use the first replacement font that matches the selection criteria. It's optional and repeatable.

The <Substitute> element has one child element, <FontName>, which specifies the source and target font mapping. The target <FontName> is required and repeatable.

The <Substitute> element has no attributes.

FontName Element

The content of the <FontName> element specifies a system or font outline file. It's optional and repeatable.

The <FontName> element has no child elements. The <Simulation> element overrides <FontName>.

The <FontName> element has the following attributes:

Attributes of the FontName element

Attribute and values	Description
select = *CDATA*	Specifies the operating system on which the `FontName` is used. The value must match one of the tokens generated by the application (for example, "Windows"). You can specify more than one selection. If any of the selections is equal to any of the selection criteria tokens, then the `FontName` matches. If this attribute is empty or not present, a `FontName` match is assumed.
encoding = *CDATA*	Indicates the encoding to be used with the specified font. Multi-byte (Unicode and PostScript CMaps) and 8-bit (cp125x, where x = 0-8; iso8859-x, where x = 1-10, 13-16) formats are supported.
simulate = `normal` | `bold` | `italic` | `bolditalic`	Specifies a font face to simulate. For example, if a font does not have a bold face font, you could specify `bold`. The default is `normal`.
simulateMode = `PDF` | `reader`	Specifies how to apply the simulation. `PDF` modifies the font within the PDF, by slanting the output for italics or using multiple registrations of characters for bold. `reader` sets a flag that tells the PDF viewer to render the simulation, which will work with standard PDF fonts. The default is `reader`. **Note** If you choose `reader`, it only applies to standard PDF fonts, which have metrics built into the PDF reader.
type = `SYS` | `TTF` | `PFA` | `PFB`	Specifies the type of font, system (SYS) or a font outline file (`.ttf`, `.pfa`, or `.pfb` file). The default is SYS.

Attributes of the FontName element (continued)

Attribute and values	Description
path = *CDATA*	Specifies the path and file name to the type of file (`.ttf`, `.pfa`, or `.pfb`) specified by *type*. If *type* = `SYS`, the path is ignored. If you specify a relative path, the path is relative to the `custom` directory.
metrics = *CDATA*	Specifies the metrics file (`.afm` or `.pfm`) to be used when *type* = `PFA` or `PFB`. If you do not specify a metrics file, Arbortext Editor searches for a `.afm` file with the same base file name as the `PFA` in the specified *path*.

Label Element

The `<Label>` element provides a title for the PDF configuration file. The `<Label>` element is optional and may only be used once.

The `<Label>` element has no child elements or attributes.

Documentation Element

The `<Documentation>` element provides content that describes the PDF configuration file.

The `<Documentation>` element has no child elements or attributes. The `<Documentation>` element is optional and may only be used once.

5

Customizing Publishing Rules

Customizing Publishing Rules

See *Publishing Rules Overview* and its related topics in the Arbortext Editor or Arbortext Publishing Engine Interactive online help for general information about creating, using, and managing publishing rules. You will need to be familiar with how publishing rules work to better understand this documentation.

Publishing Rule Output Files

Publishing Rule Output

The output of a publishing rule consists of the requested published document, referenced content, and optional logs. The output can include the following:

- The published output as specified by the publishing rule, which is one of the following, depending on the type of publishing operation:
  - a standalone file, such as PDF or HTML Help output.
  - a file with an additional directory, such as HTML File or RTF output. The directory, named *`filename`*`_files`, contains graphics and other external references if they exist. The directory is not produced if the published content has no external references.
  - an output directory containing output files and subdirectories for Web output. The directory contains the set of files comprising the content and a set of subdirectories containing supporting files, such as graphics and CSS stylesheets.
- an optional rule log that is placed in the same directory as the published output for the rule. It contains information about the publishing process, including the part of the Event Log that describes the execution of the publishing rule. The log name will be the same as the rule's output file name (including file extension) or directory name with an appended `.rulelog.xml` extension.

Publishing Rule Set Output

The output of a publishing rule set is a directory containing:

- a set of subdirectories, one for each publishing rule. Each subdirectory contains the publishing rule output according to type of operation, as described in the previous section.

an optional rule log, also described in the previous section, can be placed in the same directory as the published output for each rule being published.

- a manifest file can be placed in the rule sets's top level output directory. The rule set `manifest.xml` contains identifying information about the rule set itself, then lists each publishing rule in the rule set. For each rule, the manifest will list the rule's name, location, type and other static information. It will also list the path to the rule's output directory or output file, whether output for the rule succeeded, and how many errors and warnings were written to the Event Log (which is included in the rule log) while the rule was executing.

Publishing Rule Parameters

Publishing rule parameters are stored in the rule's definition in its rule file. The publishing rule parameters control the names of output files or directories and whether to generate log files for the publishing rule. Rule parameter values may contain several variables, defined as follows:

- *%n* is the publishing rule name
- *%u* is a discretionary sequential numbering applied by a rule set processor to ensure a directory name is unique. *%u* is an empty string when a rule runs alone.
- *%s* is the period with file extension for a published output file (whether or not it's accompanied by a directory of referenced content), as specified by the rule's `rule.outputSuffix` parameter value. If the rule produces a directory, *%s* is the empty string.

When a rule executes, the publishing rule processor substitutes the appropriate value for these variables in each rule parameter. To specify a literal `%` in your parameter value, use `%%`.

Publishing Rule Parameters

Parameter Name	Parameter Values
`rule.generateLog`	Determines whether a rule will generate a rule log `.rulelog.xml` If set to `yes`, then the rule will generate a rule log. If set to `no` (the default), then no log is generated. The rule set parameter `generateRuleLogs` can override this parameter.
`rule.outputSuffix`	Specifies the file extension used for published files. Specific types of output have default file extensions, so this parameter would be used to override the default values or supply a value for a rule with no default file extension for its output. These files extensions are used as the default values for publishing: • `.pdf` for PDF • `.htm` for HTML • `.rtf` for exporting to RTF • `.ps` for PostScript • `.chm` for HTML Help • `.xml` for publishing using XSL Publishing for Web produces an output directory.
`rule.outputTarget`	This parameter specifies the file or directory where the published output will be written for a publishing rule. The default is `%n%u%s`. It may be an absolute or relative path. The `ruleTargetPattern` and `ruleTargetOverride` for a rule set can override this parameter.

Adding a Publishing Rule Parameter

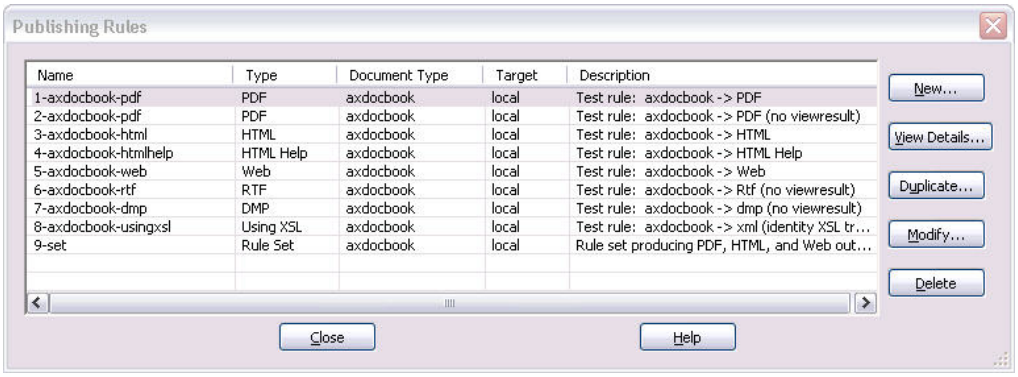
Any publishing parameter can be set using the **Advanced** tab for the publishing rule. This example uses a publishing rule from the sample publishing rules file located at:

Arbortext-path`\samples\publishingrules\sample.prcf`

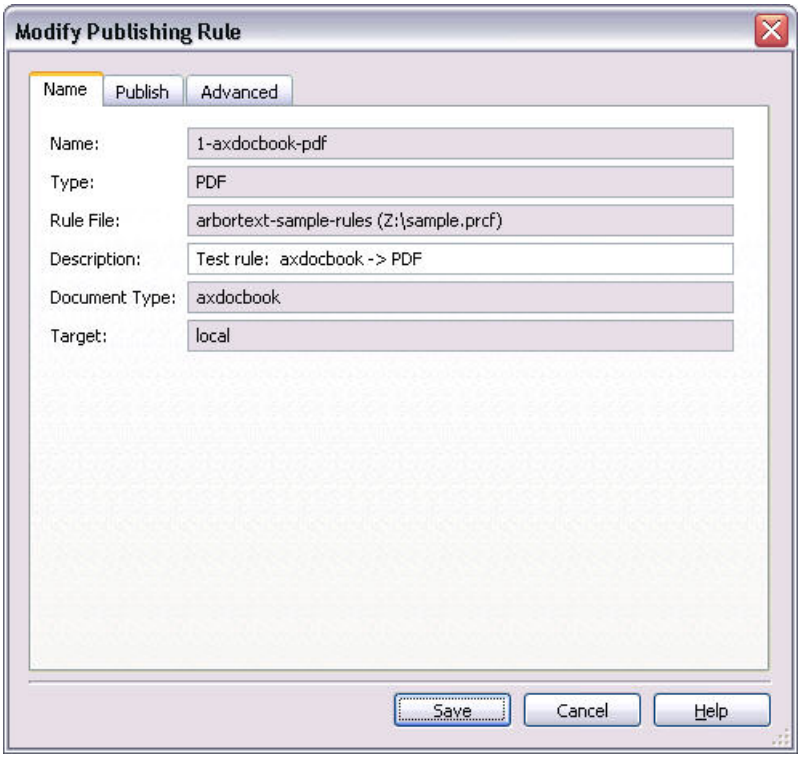
You will need to put it into a supported publishing rules directory accessible to your system, either your home directory or a `custom\publishingrules` directory, so that Arbortext Editor can find it automatically. If you are not sure of the location of your home directory, you can choose **Help** ▸ **Session** and find **Home directory** in the list.

The following example shows how to add the `rule.generateLog` publishing rule parameter to the **Advanced** tab for a sample publishing rule.

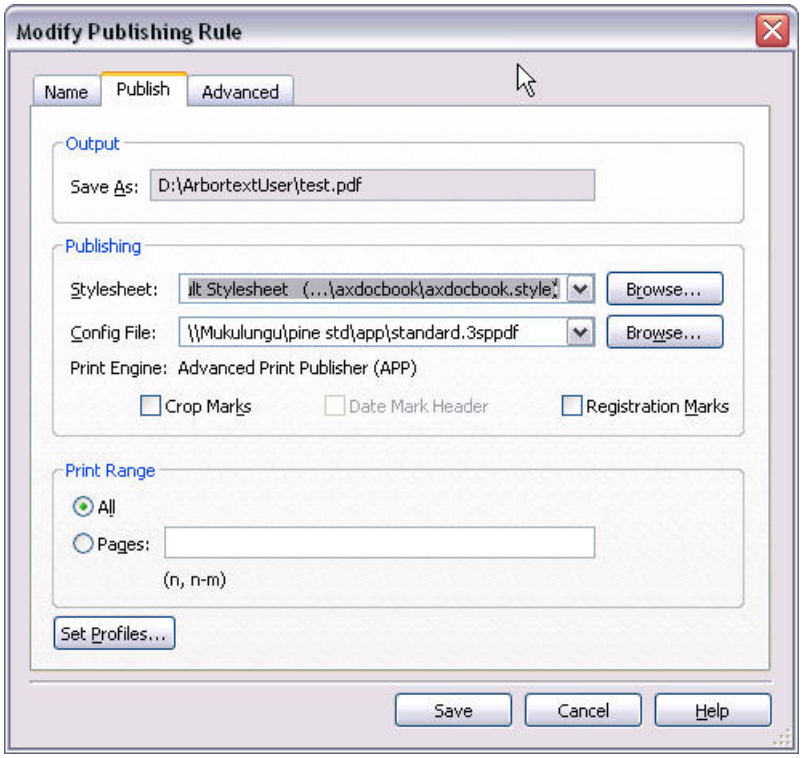
1. Open **Tools** ▸ **Administrative Tools** ▸ **Publishing Rules** and choose a rule.



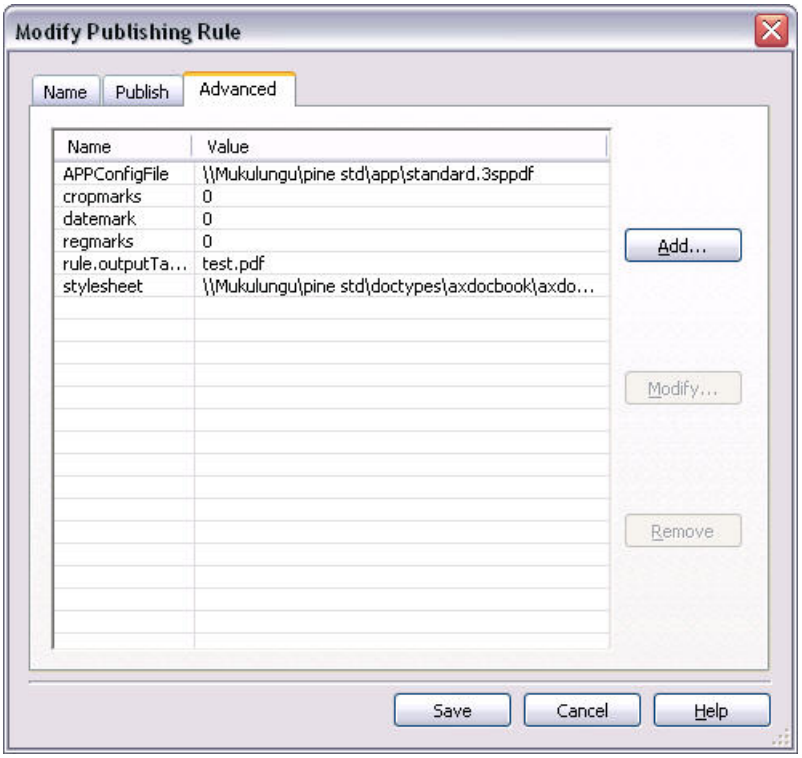
2. Choose **Modify** to open **Modify Publishing Rule**.
3. On the **Name** tab, all rule or rule set identification information is displayed.



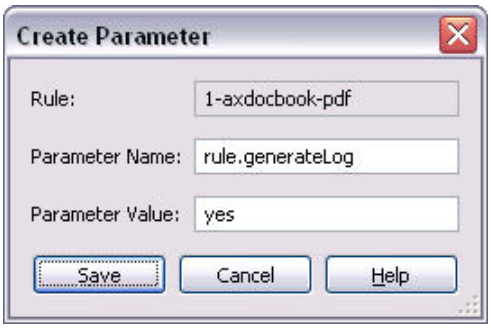
4. Choose the **Publish** tab to view the publishing settings for the rule.



5. Choose the **Advanced** tab. The list displays all the advanced parameters and their values for the rule. Some of the parameters already selected will be listed here. If some parameters are using defaults, they may not be listed.



6. Choose **Add** to add the `rule.generateLog` publishing rule parameter.



   Click **Save** to save the `rule.generateLog` parameter to the list of **Advanced** parameters.

7. Click **Save** again to save the parameter with the rule definition. The next time you publish with this rule, the rule log file will be generated along with the PDF file.

Publishing Rule Set Parameters

Rule set parameters are stored in the rule set's definition in its rule file. The publishing rule set parameters control how paths are handled, directory naming conventions for multiple rules, and generating logs and manifest files. Rule set parameter values may contain several variables, defined as follows:

- *%n* is the publishing rule name
- *%u* is a discretionary sequential numbering applied by a rule set processor to ensure a directory name is unique. *%u* is an empty string when a rule runs alone.
- *%s* is the period with file extension for a published output file (whether or not it's accompanied by a directory of referenced content), as specified by the rule's `rule.outputSuffix` parameter value. If the rule produces a directory, *%s* is the empty string.

When a rule set is executed, the publishing rule processor substitutes the appropriate value for these variables in each rule set parameter. To specify a literal `%` in your parameter value, use `%%`.

Rule Set Parameters

Parameter Name	Parameter Values
`absoluteManifestPaths`	Determines whether paths to output files and directories are specified in the manifest file as absolute or relative paths, if a manifest file is generated (see `generateManifest`). If set to `on`, paths to output files and output directories in the rule set manifest will be specified as absolute paths. If set to `off` (the default), paths to output files or directories in the rule set manifest will be specified as relative paths. The paths will be relative to the top level directory for the rule set. Paths to files or directories that are outside the rule set's output directory will always be specified as absolute paths.
`generateManifest`	Determines whether a manifest file is generated for the output of a rule set. If set to `yes`, `manifest.xml` is written to the top level rule set output directory, which lists the results of the rule set execution.

Rule Set Parameters (continued)

Parameter Name	Parameter Values
	If set to `no` (the default), a manifest file is not generated.
`generateRuleLogs`	Determines whether each rule in a rule set will generate a rule log. If set to `yes`, then every rule in the rule set will generate a rule log. If set to `no` (the default), then each rule in the set will generate a log according to the value of its `rule.generateLog` parameter.
`outputMode`	This parameter has the value `separate`, which means the output for each rule in a rule set is placed in a separate directory.
`outputModePattern`	Specifies the string to be used in constructing an output directory for a rule in a rule set. The default is `rule-%n%u`. For example, if set to `output%u`, each output directory would be named `ouput`*n* (`output0`, `output1`, `output2` and so on).
`outputTarget`	This parameter specifies the directory where the rule set output will be written. The default is `ruleset-%n`, where *%n* is the rule set name. It may be an absolute or relative path.

Rule Set Parameters (continued)

Parameter Name	Parameter Values
`ruleTargetOverride`	Determines whether to override the `rule.outputTarget` parameter of each rule in the rule set, and replace it with the value of `ruleTargetPattern`. If set to `yes`, then each rule parameter value for `rule.outputTarget` is replaced by the value of the rule set parameter `ruleTargetPattern`. If set to `no` (the default), then the `rule.outputTarget` for the rule is obeyed. For example, if a rule is named `rule23`, and the rule set specifies `ruleTargetPattern` as `output-%n`, specifying `yes` means that `output-rule23` will be substituted as the output directory.
`ruleTargetPattern`	Specifies the value to replace each rule's `rule.outputTarget` parameter, if `ruleTargetOverride` is also set to `yes`. The default is `%n%u%s`. For example, if a rule is named `rule23`, and `ruleTargetPattern` is set to `output-%n` and `ruleTargetOverride` is set to `yes`, `output-rule23` will be substituted as the output directory.

Adding a Publishing Rule Set Parameter

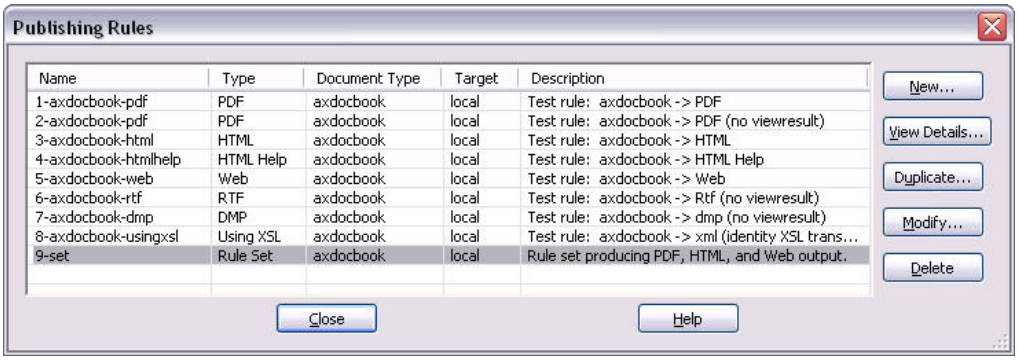
This example uses the publishing rule set from the sample publishing rules file located at:

Arbortext-path`\samples\publishingrules\sample.prcf`

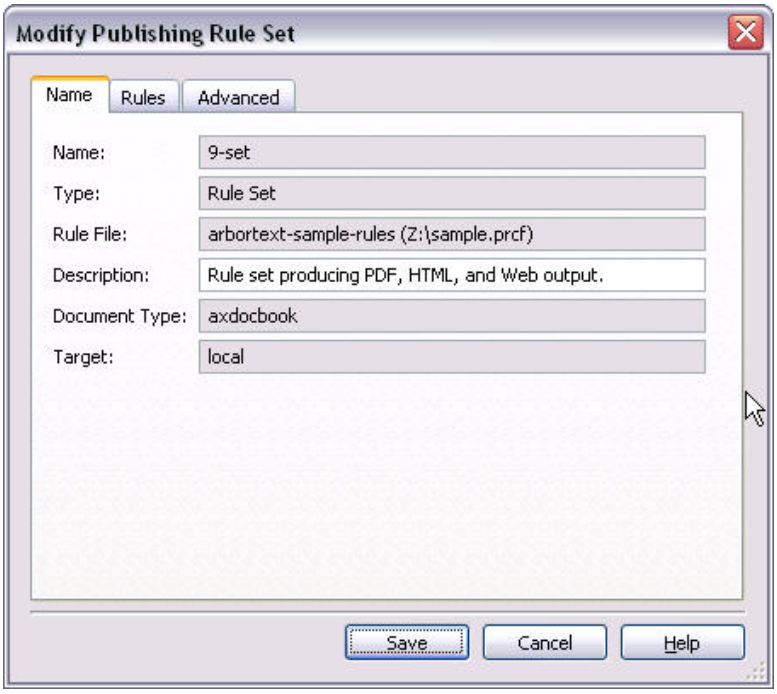
You will need to put it into a supported publishing rules directory accessible to your system, either your home directory or a `custom\publishingrules` directory, so that Arbortext Editor can find it automatically. If you are not sure of the location of your home directory, you can choose **Help ▸ Session** and find **Home directory** in the list.

The following example shows how to add the `absoluteManifestPaths` rule set parameter to the **Advanced** tab for the sample publishing rule set.

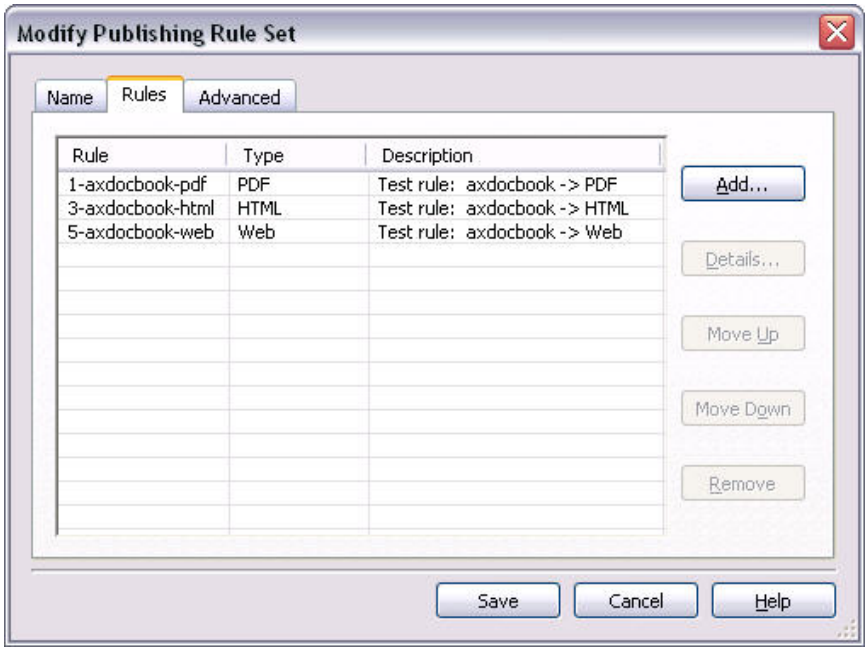
1. Open **Tools ▸ Administrative Tools ▸ Publishing Rules** and choose the rule set.



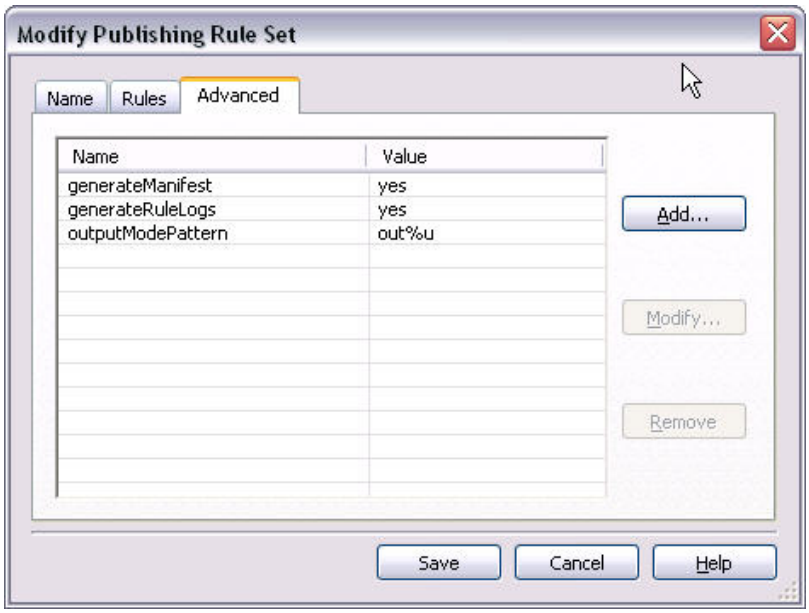
2. Choose **Modify** to open **Modify Publishing Rule Set**.
3. On the **Name** tab, all rule or rule set identification information is displayed.



4. Choose the **Rules** tab to view the list of publishing rules in the rule set.



5. Choose the **Advanced** tab. The list displays the advanced parameters and their values for the rule. If some parameters are using defaults, they may not be listed.



6. Choose **Add** to add the `absoluteManifestPaths` publishing rule set parameter.

Click **Save** to save the `absoluteManifestPaths` parameter to the list of **Advanced** parameters.

7. Click **Save** again to save the parameter with the rule definition. The next time you publish with this rule set, the manifest file will be generated using absolute paths.

Overriding Rule Parameters

A rule set can override a rule parameter by specifying a parameter with the following syntax:
`override:`*`file`*`:`*`rule`*`:`*`parameter=override-value`*

- *file* (optional)

  Specifies the rule file Unique ID assigned to the rule file when it was created.

  If omitted, the override will apply to the first publishing rule matching the *rule* specification.

- *rule*

  Specifies the rule name.

- *parameter*

  Specifies the parameter name that will have its value replaced by the *override-value*.

- *override-value*

  Specifies the value that will replace the value for *parameter* in the rule.

A rule set can override the same parameter for every rule it contains by specifying a rule set parameter:
`override:all:`*`parameter=override-value`*

- *parameter*

  Specifies the parameter name that will have its value replaced by the *override-value* for every rule in the rule set.

- *override-value*

Specifies the value that will replace the value for *parameter* in every rule.

For example, `override:all` could specify the `debug` parameter for every rule in the rule set by setting `override-all:debug` to the value `1`.

Rule and Rule Set Error Handling

Error handling for rules and rule sets:

- Any rule parameter covered in the preceding documentation that has an invalid value will be logged in the rule log as a Warning message. Publishing processing will use the parameter's default value instead.
- Any rule parameter that is not recognized by the rule processor will be ignored, presuming the intent is to pass it on to the publishing framework.
- Any rule set parameter covered in the preceding documentation that has an invalid value will be logged in the manifest as a Warning message. Publishing rule processing will use the default value instead.
- Any rule set parameter not covered in the preceding documentation (not recognized by the rule processor) will be logged in the rule set manifest (if one is generated) as a Warning message.

Arbortext Publishing Engine Document Conversion

If you are using a client application and the `f=convert` function to send publishing requests to the Arbortext PE server, you can control the rule set parameters to be used during publishing on the Arbortext PE server. To use the rule set parameters as defined in the rule file, specify `use-ruleset-parameters=yes` on the HTTP `f=convert` request.

If you specify `use-ruleset-parameters=yes`, then Arbortext Publishing Engine will use the parameter values specified in the rule set definition in the rule file. If a parameter required for publishing processing is not specified in the rule file, the default value will be used.

If you specify `use-ruleset-parameters=no` (the default), then Arbortext Publishing Engine will ignore all rule set parameters in a rule file for `f=convert` requests, and it will use all default values for rule set parameters.

For more information on using `f=convert`, see the *Arbortext Publishing Engine Document Conversion* chapter of the *Programmer's Guide to Arbortext Publishing Engine*.

6

Working with XUI (XML-based User Interface) Dialog Boxes

XUI Overview

The Arbortext XUI (XML-based User Interface) technology lets an application programmer create, display, manipulate, and modify dialog boxes in real time by writing and modifying XML documents. All aspects of a dialog box, including controls, layout, and event listeners, can be stored in a single XUI XML document. The document type for XUI XML documents is *`Arbortext-path`*`\doctypes\xui\xui.dtd`. The Arbortext XUI technology is influenced by the Mozilla XML User Interface Language (XUL) and its ongoing development.

Each control in a XUI dialog box is associated with an element in an XML document. For example, to add a check box control to the dialog box, insert a `checkbox` element in to the XUI document. A control's properties are represented by the attributes of the element. For example, the `label` attribute of the `checkbox` element specifies the label of the check box control in the dialog box. The `checked` attribute of the element represents the current status of the check box control.

The layout of XUI dialog boxes is automatically managed by Arbortext Editor. The type of the layout and the properties of the layout are specified in the XUI XML document by inserting layout related elements and attributes. XUI layout algorithms arrange layout by considering the size of each control and ensuring that the dialog box is displayed in a balanced manner after resizing.

Arbortext Editor makes use of W3C XML Events to monitor the activities in the dialog box. The W3C XML Events specification can be found at www.w3.org/TR/2001/WD-xml-events-20011026. Event listening scripts can be inserted in the XUI XML document as the content of an element representing an event listener.

A set of AOM interfaces manipulate XUI dialog boxes. For example, the **Window** interface represents the frame window of the XUI dialog box. You can use the `hide` method to hide the dialog box, the `setTitle` method to set the window's title, and so on. Refer to Manipulating XUI Dialog Boxes using the AOM on page 126 for more details.

XUI dialog boxes can be displayed as standard dialog boxes (overlaying an application or document), and they can be displayed embedded in a document in the Edit pane in Arbortext Editor. Refer to Embedding XUI Dialog Box Controls in a Document on page 145 for details on displaying XUI dialog boxes within documents. Standard dialog boxes with docking enabled can be dragged into a Arbortext Editor edit window and docked on one edge of the edit window. Dialog boxes with docking enabled can contain all controls that are allowed in a standard dialog box except for toolbars and menubars.

Defining the Dialog Box

You can use Arbortext Architect to interactively define and modify XUI dialog boxes. With Arbortext Architect open, choose **Edit ▸ XUI**. Open your XUI XML document. (A sample file is at *Arbortext-path*\doctypes\xui\demo.xml.)

With Arbortext Architect displaying the XUI XML document, choose **Tools ▸ View Dialog**. The XUI dialog box will be displayed.

Changing values and settings in the XML document will be reflected immediately in the dialog box. Likewise, modifying controls in the dialog box will cause immediate updates to the XML document.

You should save your custom XUI files in the *Arbortext-path*\custom\dialogs directory.

Displaying the Dialog Box using the AOM

The AOM `Application.createDialogFromFile()` method creates XUI dynamic dialog boxes.

To display a XUI dialog box (in this example, the dialog box is defined in the XML file `C:\Project\find.xml`), you can run the following JavaScript statements:

```
var dialog = Application.createDialogFromFile("c:\Project\find.xml");
dialog.show();
```

Another method, `Application.createDialogFromDocument()`, creates a XUI dynamic dialog box from an existing DOM **Document**.

As when defining the XUI dialog box, value and setting changes in the XML document will be reflected immediately in the dialog box. Likewise, modifying controls in the dialog box will cause immediate updates to the XML document.

To suspend immediate updates, set the value of **View.suspendupdates** to `TRUE`.

Describing Dialog Box Controls

Dialog boxes and their controls are defined using the XUI elements described in XUI Element Reference on page 148. Refer to that section for detailed descriptions of each element and its attributes.

For example, the following XUI markup defines this dialog box:

Dialog box with `label`, `textbox`, and two `button` controls.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window title="Find">
 <label label="&Find what:"/>
 <textbox multiline="false">
 <value></value>
 </textbox>
 <button label="Find &Next" type="accept"></button>
 <button label="Cancel" type="cancel"></button>
</window>
```

The content of the `value` element will become the text the user enters into the text box in the dialog box. A button with the type `cancel` is activated when the **Esc** key is pressed. A button with the type `accept` is the default button and is activated when the **ENTER** key is pressed.

Specifying Dialog Box Layout

XUI supports three layout models: Box, Grid, and Morph.

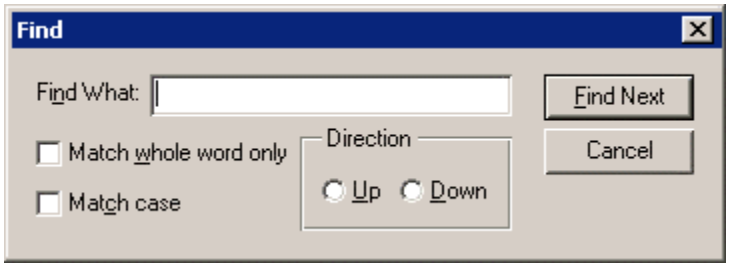
Layout models are specified by using their corresponding elements.

Layout Model	XUI Element
Box layout	box
Grid layout	grid
Morph layout	morph

Box Layout

The Box layout model divides a portion of the dialog box into a series of nested containers using the `<box>` element. Each container is oriented horizontally or vertically. Controls inside each container are listed in order according to the orientation of the container.

For example, the following dialog box can be expressed by using seven containers:



Dialog box with the following controls.

Box 1 contains the Find What label and its text field.

Box 2 contains the two check boxes.

Container 3 is the Direction radio group.

Box 4 contains the Find Next and Cancel buttons.

Box 5 contains box 2 and container 3.

Box 6 contains box 1 and box 5.

Container 7 is the root (<window>), which contains box 4 and box 6.

This dialog box can be described using XUI as follows:

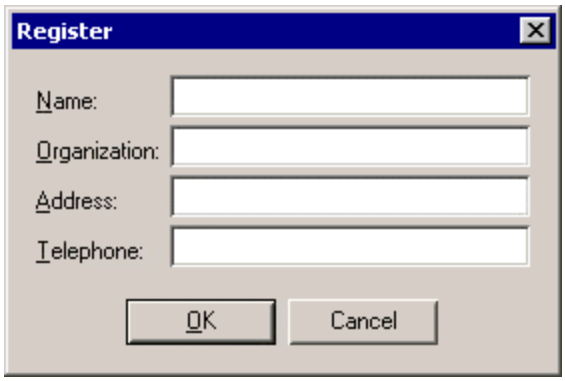
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window orient="horizontal" focus="findtext" modal="false" title="Find">
 <box orient="vertical">
 <box orient="horizontal">
 <label label="Fi&nd What:"/>
 <textbox id="findtext">
 <value></value>
 </textbox>
 </box>
 <box orient="horizontal">
 <box orient="vertical">
 <checkbox label="Match &whole word only"></checkbox>
 <checkbox label="Match &case"></checkbox>
 </box>
 <radiogroup label="Direction" resize="both" orient="horizontal">
 <radio label="&Up"></radio>
 <radio label="&Down"></radio>
 </radiogroup>
 </box>
 </box>
 <spacer resize="none" width="4"/>
 <box orient="vertical" pack="start">
 <button label="&Find Next" type="accept"></button>
 <button label="Cancel" type="cancel"></button>
 </box>
</window>
```

The <box> element has attributes for customizing the layout. For example, the attribute *pack* specifies how extra space should be distributed. The attribute *orient* specifies the baseline of the controls in a box. Refer to <box> Element on page 149 for a detailed description of the <box> element.

Grid Layout

The Grid layout model lets you display a group of controls in rows and columns using the <grid> element.

For example, the following dialog box uses the Grid layout.



Dialog box with a Grid layout.

This dialog box has two columns and four rows. It is described using XUI as follows:

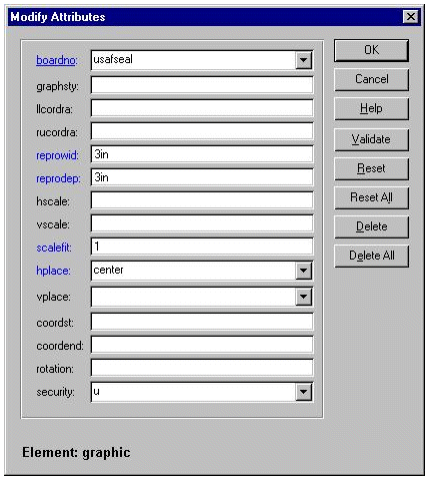
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window orient="vertical" modal="true" title="Register">
 <grid columns="2">
 <label label="&Name:"/>
 <textbox width="180">
 <value></value> </textbox>
 <label label="&Organization:"/>
 <textbox>
 <value></value>
 </textbox>
 <label label="&Address:"/>
 <textbox>
 <value></value>
 </textbox>
 <label label="&Telephone:"/>
 <textbox>
 <value></value>
 </textbox>
 </grid>
 <spacer height="8" resize="none"/>
 <box orient="horizontal" pack="center">
 <button label="&OK" type="accept"/>
 <button label="Cancel" type="cancel"/>
 </box>
</window>
```

Refer to <grid> Element on page 165 for a detailed description of the `<grid>` element.

Morph Layout

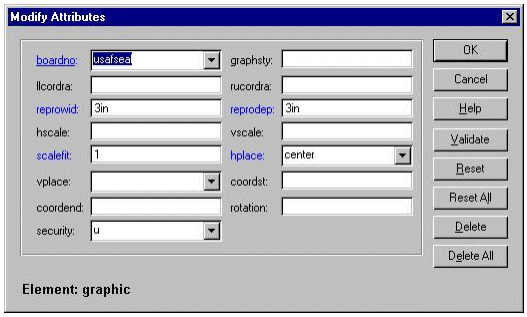
The Morph layout uses the `<morph>` element to create a dialog box that dynamically adjusts the layout of its contents. This makes the Morph layout very useful for displaying a dialog box with an varying set of controls based on context or containing a large number of controls.

Consider a dialog box with a Morph layout that initially displays its contents in the same manner as a Grid layout:



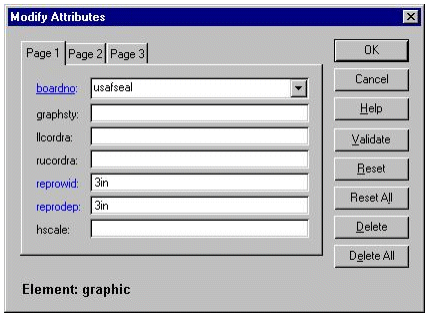
A dialog box with a Morph layout with a single column of `label` and related `textbox` controls.

As the dialog box is resized, the layout may change to include additional columns. For example:



A dialog box with a Morph layout with two columns of `label` and related `textbox` controls.

When the dialog box is resized so small that all of the elements can no longer be displayed, the layout changes to a tabbed box. For example:



A dialog box with a Morph layout with tabs, each with a single column of `label` and related `textbox` controls.

This Morph dialog box is described in XML as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window title="Modify Attribute">
 <box orient="horizontal">
 <box orient="vertical">
 <grid morph="true">
 <label label="boardno:"/>
 <combobox dropdown="true">
 <listitem label="usafseal"/>
 </combobox>
 <label label="graphsty:"/> <textbox/>
 <label label="llcordra:"/> <textbox/>
 <label label="rucordra:"/> <textbox/>
 <label label="reprowid:"/> <textbox/>
 <label label="reprode:p"/> <textbox/>
 <label label="hscale:"/> <textbox/>
 <label label="vscale:"> <textbox/>
 <label label="scalefit:"/> <textbox/>
 <label label="hplace:"/>
 <combobox dropdown="true">
 <listitem label="center"/>
 <listitem label="left"/>
 <listitem label="none"/>
 <listitem label="right"/>
 </combobox>
 <label label="vplace:"/>
 <combobox dropdown="true">
 <listitem label="bottom"/>
 <listitem label="middle"/>
 <listitem label="none"/>
 <listitem label="top"/>
 </combobox>
 <label label="coordst:"/> <textbox/>
 <label label="coordend:"/> <textbox/>
 <label label="rotation:"/> <textbox/>
 <label label="security:"/>
 <combobox dropdown="true">
```

```
 <listitem label="c"/>
 <listitem label="s"/>
 <listitem label="u"/>
 </combobox>
 </grid>
 <description label="Element: graphic"/>
 </box>
 <box orient="vertical">
 <button label="OK"/>
 <button label="Cancel"/>
 <button label="Help"/>
 <button label="Validate"/>
 <button label="Reset"/>
 <button label="Reset All"/>
 <button label="Delete"/>
 <button label="Delete All"/>
 </box>
 </box>
</window>
```

The Arbortext Editor **Modify Attributes** dialog box is an example of a Morph layout. While editing a document with Arbortext Editor, choose **Edit ▸ Modify Attributes**. The **Modify Attributes** dialog box displays all of the attributes for the current element. If you resize the dialog box, the layout changes to accommodate the controls.

Refer to <morph> Element on page 178 for a detailed description of the `<morph>` element.

Specifying Event Listeners

Since a XUI dialog box is synchronized with its XUI XML document, you can register DOM events in the document to monitor activities in the dialog box. For example, when a check box is selected, the `checked` attribute of the element will change to `true`. If you register a `DOMAttrModified` event on the `checkbox` element, you will be informed whenever the check status is changed in the dialog box.

XUI extends the `DOMActivate` event to be dispatched when the status of a control is changed. For example, a `DOMActivate` event will be dispatched to the affected element whenever a button is clicked, a check box is selected, an item in a list box is selected, the content of a text box is changed, and so on. Therefore, you can register a `DOMActivate` event listener on a `button` element to execute the necessary routines when a button is pushed.

XUI makes use of W3C XML Events. You can register event listeners in the XUI document as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window>
```

```
 <button label="OK">
 <script type="application/x-javascript" ev:event="DOMActivate">
 Application.alert("The OK button is selected");
 </script>
 </button>
</window>
```

This document declares the XML Events namespace in the `window` element. The `script` element contains the script that will be executed when the event specified by the *ev:event* attribute is dispatched. Therefore, the `script` element registers a `DOMActivate` event listener on the `button` element, and the body of the event listener is the content of the `script` element. When the **OK** button is pushed, the JavaScript engine will display a message box with the message "`The OK button is selected`".

The *type* attribute of the `script` element specifies the type of the script. XUI supports JavaScript (Rhino), JScript (Microsoft), and VBScript. Refer to <script> Element on page 185 for a list of valid script types.

Note

Be aware that XUI can make use W3C UIEvent events, XUI cannot make use of MouseEvent events, a subclass of UIEvent. MouseEvent is currently supported only for documents in the Arbortext Editor edit pane.

In addition to DOM Events, you can register AOM `WindowEvent` events on the `window` element. The `WindowEvent` module has the following event types:

Event type	Time to dispatch
`WindowLoad`	Before the dialog box is loaded on the screen.
`WindowClosing`	After the dialog box title bar's close button is selected.
`WindowClosed`	After the dialog box is dismissed.
`WindowCreated`	After the dialog box is created. (Listeners can only be added to the **Application** object.)
`WindowActivated`	After the dialog box gains focus.
`WindowDeactivated`	After the dialog box loses focus.
`WindowMinimized`	After the dialog box is minimized.
`WindowRestored`	After the dialog box is restored from being minimized.

The following examples show the three ways to register an event listener:

Example

Use the listener to associate an event handler with its observer

In this example, the *usage* attribute <script> is set to indirect. Doing so disassociates the <script> element from its parent <window> element. If *usage* is set to direct (the default), the observer of the handler is the handler's parent element.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window>
 <button id="ok" label="OK"/>
 <script id="select" usage="indirect" type="application/x-javascript">
 Application.alert("The OK button is selected");
 </script>
 <ev:listener event="DOMActivate" observer="ok" handler="#select"/>
</window>
```

Example

Attach attributes directly to the observer element

As in the previous example, the *usage* attribute of <script> is set to indirect, disassociating the <script> element from its parent <window> element.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window>
 <button label="OK" ev:event="DOMActivate" ev:handler="#select"/>
 <script id="select" usage="indirect" type="application/x-javascript">
 Application.alert("The OK button is selected");
 </script>
</window>
```

Example

Attach attributes directly to the handler element

The observer is the parent element of the handler. This example illustrates the simplest way to register an event listener.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN" "xui.dtd">
<window>
 <button label="OK">
 <script type="application/x-javascript" ev:event="DOMActivate">
 Application.alert("The OK button is selected");
 </script>
 </button>
</window>
```

Returning Values from Dialog Boxes

Because the XUI document is updated when any values or states change in the control it defines, the value of the control can be obtained by evaluating the content of the XUI document. Based on the XUI DTD, each control type (textbox, checkbox, and so on) potentially stores its value in a different manner in the document.

For example, a `<textbox>` element stores its value as the content of its child `<value>` element. A `<combobox>` element stores it's value in its *value* attribute. Refer to XUI Element Reference on page 148 (or the XUI DTD) for details on each control.

The following examples return the current values of different dialog box controls.

Example

Returning a value from a textbox control

The value of a `<textbox>` element is stored as the content of the child `<value>` element. The following example displays the value in a message window when the **OK** button is selected.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN"
 "xui.dtd">
<window orient="horizontal" align="center" modal="false" title="Value Test">
 <label label="User ID"/>
 <textbox id="userid">
 <value></value>
 </textbox>
 <button label="OK" type="accept">
 <script type="application/x-javascript" ev:event="domactivate">
 var document = Application.event.target.ownerDocument;
 var textnode = document.getElementById("userid").firstChild.firstChild;
 if (textnode) {
 Application.alert(textnode.nodeValue);
}
 else {
 Application.alert("[ EMPTY]");
}
 var dialog = Application.event.view.window;
 dialog.close();
 </script>
 </button>
</window>
```

Example

Returning a value from a combobox control

The value of a <combobox> is stored in its *value* attribute. The following example displays the value in a message window when the **OK** button is selected.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN"
 "xui.dtd">
<window orient="horizontal" align="center" modal="false" title="Value Test">
 <label label="Choose a color"/>
 <combobox id="color" value="Red" type="dropdownlist">
 <listitem label="Red"/>
 <listitem label="Orange"/>
 <listitem label="Yellow"/>
 <listitem label="Green"/>
 <listitem label="Blue"/>
 </combobox>
 <button label="OK" type="accept">
 <script type="application/x-javascript" ev:event="domactivate">
 var document = Application.event.target.ownerDocument;
 var value = document.getElementById("color").getAttribute("value");
 Application.alert(value);
 var dialog = Application.event.view.window;
 dialog.close();
 </script>
 </button>
</window>
```

Example

Returning a value from a multiple-selection listbox control

You can select 1 or more values in a multiple-selection <listbox> control. The following example displays each selected value in a message window when the **OK** button is selected.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN"
 "xui.dtd">
<window orient="horizontal" modal="false" title="Value Test">
 <label label="Choose Colors"/>
 <listbox id="color" width="100" height="100" value="Red" type="multiple">
 <listitem label="Red" selected="true"/>
 <listitem label="Orange"/>
 <listitem label="Yellow" selected="true"/>
 <listitem label="Green"/>
 <listitem label="Blue"/>
```

```
 </listbox>
 <button label="OK" type="accept">
 <script type="application/x-javascript" ev:event="domactivate">
 var document = Application.event.target.ownerDocument;
 var listbox = document.getElementById("color");
 var nodelist = listbox.getElementsByAttribute("selected", "true", 1);
 for (var i = 0; i < nodelist.length; i++) {
 Application.alert(nodelist.item(i).getAttribute("label"));
}
 var dialog = Application.event.view.window;
 dialog.close();
 </script>
 </button>
</window>
```

For more information on working with events, refer to the Events chapter in the *Programmer's Reference*.

Manipulating XUI Dialog Boxes using the AOM

AOM **Window** and **View** interfaces work on both XUI dynamic dialog boxes and Arbortext Editor Edit windows. A XUI **Window** object can be obtained when the dialog box is created. For example, the following JavaScript statement returns an AOM **Window** object.

```
var window = Application.createDialogFromFile("C:\find.xml");
```

An Edit **Window** object can be obtained by converting an ACL window ID to its corresponding AOM **Window** object. For example:

```
var window = Acl.getWindow(5);
```

Whether or not the window with focus is a XUI dialog box or an Edit window, you can use the following JavaScript statement to get its **Window** object.

```
var window = Application.activeWindow;
```

Once you have the **Window** object, you can call the methods of the **Window** interface to manipulate the window. For example, the following JavaScript code moves the active window to a specific location:

```
Application.activeWindow.moveTo(200, 200);
```

You can find a complete list of methods for the **Window** interface in the Window interface chapter of the *Programmer's Reference*.

Customizing the Preferences Dialog Box

When Arbortext Editor loads the **Preferences** dialog box, it scans the dialog search path looking for a file called `pref_exts.xml`. If any are found, they will be processed in order. Arbortext Editor opens the file, parses it, and adds the

new preferences categories to the XUI document which controls the **Preferences** panel. You can use the `pref_exts.xml` file to extend the **Preferences** dialog box for your custom application.

When Arbortext Editor launches, it finds any `pref_exts.xml` files and adds one or more new items to the list under **Category** as specified in the file. When the user chooses the new category, its relevant preference settings are displayed adjacent to the right as specified by the designed by the `pref_exts.xml` file.

The `pref_exts.xml` file must be placed in a location where Arbortext Editor can locate it at startup, such as the `custom\dialogs` directory or in a `dialogs` subdirectory of a custom application running under the `application` directory. Both of these locations are part of the `set dialogspath` by default.

A sample Preference Dialog Extension that you can use as a starter for your custom application is in the *`Arbortext-path`*`\samples\xui\preferences` directory. This example uses the `custom` directory structure and contains several files to be placed within its subdirectories. The `pref_exts.xml` file contains comments that describe the rules and guidelines for using this capability.

XUI Dialog Boxes and ACL

All XUI dialog boxes support `dlgitem_set()`, `dlgitem_get()`, and all variations of `dlgitem_set_`*`xxx`*`()` and `dlgitem_get_`*`xxx`*`()` ACL functions. The *id* attribute of the element in the XUI document is the dialog box control name. If a control has no *id* specified, the control has no dialog box control name and cannot be used with `dlgitem_set()`.

For example, if the XUI document contains the following `checkbox` element:

```
<checkbox id="full" label="Full Menus"/>
```

you can use `dlgitem_set(win, 'full', 'VALUE', 1)` to mark the check box as checked.

When using any of the `dlgitem_set_`*`xxx`*`()` functions, the value displayed in the dialog box control will be updated on-screen. However, the underlying XUI document will not change. That is, the value in the XUI document will not be updated by the `dlgitem_set_`*`xxx`*`()` function.

Working with Images

Images can be used in many areas of XUI dialog boxes, such as in check boxes, on menus, and on buttons. (However, images are not available on submenu items.) Images are first defined using `image` or `imagelist` elements. The images are then referenced using the *image* attribute of the element displaying the image.

The following example uses the `image` element to define the image `c:\temp\logo1.jpg` and display it on a button.

```
<window>
 <imagegroup>
 <image id="imageLogo" path="c:\temp\logo1.jpg"/>
 </imagegroup>
 ...
 <button image="imageLogo"></button>
 ...
</window>
```

When deploying XUI controls, absolute paths to images may be restrictive. For XUI controls referencing images by name only, Arbortext Editor will first search the paths defined with the `set dialogspath` command option (and Advanced Preference). If the image is not found, Arbortext Editor will search the paths defined with `set graphicspath`. Images with relative paths will also be searched for in directories relative to the paths defined by `set dialogspath` and `set graphicspath`.

Another approach to working with images is to use the `imagelist` element to specify that a single graphic file contains multiple images of identical widths and heights.

For example, if *path* specifies a graphic file with a width of 48 and height of 16, and *imagewidth* is set to a value of 16, `imagelist` will contain 3 `image` elements, each defining an image 16 pixels wide by 16 pixels high.

The following example uses the `imagelist` element to define three images in `tool_icons.jpg` and displays them in a list box.

```
<window>
 <imagegroup>
 <imagelist path="tool_icons.bmp" imagewidth="16">
 <image id="imageIconCopy"/>
 <image id="imageIconCut"/>
 <image id="imageIconPaste"/>
 </imagelist>
 </imagegroup>
 ...
 <listbox>
 <listitem image="imageIconCopy">
 <listitem image="imageIconCut">
 <listitem image="imageIconpaste">
 </listbox>
 ...
</window>
```

If a menu item has the same command as a toolbar button, then the button's image is displayed next to the item on the menu. For example, if a toolbar `button` element has *command* set to `FileNew` and *image* set to `image.jpg`, and a `menuitem` element also has *command* set to `FileNew`, the image `image.jpg` will be displayed next to the menu item.

Working with Menus

You can create menus using the <menubar> and <menuitem> elements. Menus can be create on menubars, as shortcut menus, and as dropdown menus.

A <menuitem> can execute the ACL command specified in its *command* parameter, or the actions defined in its child <script> element using menuselected and menupost event listeners. The menuselected listener can contain the command for the menu item. The menupost listener can disable the menu item when certain conditions are met. Listeners can also be registered for the same events on items within a shortcut or dropdown menu. The posting of a menu for either a menubar or a control will cause the event ITEM_POSTMENU to be dispatched. The selection of an item will cause the existing ITEM_CHANGED event to be dispatched.

Menus on Menubars

Menu bar menus can contain any combination of the following types of menu items:

- Button menu item — Appears as a standard menu selection with an optional graphic to the left of its label. This is the only type of menu item that can have child <menuitem> elements.
- Menu separator — A line separating adjacent items.
- Toggle menu item — When activated, the item will display a check mark to the left of its label.
- Radio menu item — The menu item is displayed as a radio button.

Example

Sample menu bar

The following example creates a menu named **Custom** that contains the following menu items:

- A **Find** menu item that uses ACL to open the **Find/Replace** dialog box
- An **Options** menu with the following menu items:
    - A **Cut** menu item that is disabled if no content in the document is selected. If content is selected, the **Cut** menu item becomes available and, when chosen, will cut the selected content to the clipboard.
    - A **Full Menus** menu item that toggles the available Arbortext Editor menus

between standard and full menus. **Full Menus** has a check mark next to it when enabled.

- A **My ACL Function** menu item that displays a response dialog box. This item is a template for inserting your own function.

```
<window width="150" height="40">
 <menubar>
 <menuitem label="Custom">
 <menuitem label="Find" command="FindReplace"></menuitem>
 <menuitem label="Options">
 <menuitem label="Cut" shortcut="Ctrl+X" command="EditCut">
 <script type="application/x-javascript" ev:event="menupost">
 if (Application.activeDocument.textSelection.collapsed == true) {
 Application.event.target.enabled=false;
}
 else {
 Application.event.target.enabled=true;
}</script>
 </menuitem>
 <menuitem label="Full Menus" type="toggle">
 <script type="application/x-javascript" ev:event="menupost">
 if (Application.getOption("fullmenus") == "on") {
 Application.event.target.checked=true;
}
 else {
 Application.event.target.checked=false;
}
 </script>
 <script type="application/x-javascript" ev:event="menuselected">
 if (Application.event.target.checked == true) {
 Application.setOption("fullmenus", "off");
}
 else {
 Application.setOption("fullmenus", "on");
}
 </script>
 </menuitem>
 </menuitem>
 <menuitem label="-" type="separator">
 </menuitem>
 <menuitem label="My ACL function"
 command="response('Put your ACL function call here')">
 </menuitem>
 </menuitem>
 </menubar>
</window>
```

Shortcut and Dropdown Menus

You can create shortcut and dropdown menus in the following situations:

- Shortcut menu assigned to a tree control and customized in context for its nodes.
- Shortcut menu assigned to a table control and customized in context for its cells and rows.
- Dropdown menus for button controls.
- Dropdown menus for toolbar buttons controls.

When implemented, shortcut menus are displayed by placing the mouse cursor over the control and right-clicking. Shortcut menus are also displayed by pressing the Application key when the control has focus. Dropdown menus are displayed by left-clicking on a control and by activating a control when it has focus.

The `Acl` event notification API supports menus within XUI dialog boxes. Menus can be specified in menu configuration files as well, but the event management for these menus does not use the `Acl` event enhancements. Instead, event management is specified within the configuration files.

An example application of XUI menus with menu bars, controls, and tool bars is provided in the following directory:
Arbortext-path`\samples\XUI\xuimenusample`

Working with Toolbars

XUI provides limited toolbar support. You can create toolbars on dialog boxes using the `<toolbargroup>` and `<toolbar>` elements. A toolbar is made up of a collection of toolbar buttons defined with the `<button>` and `<checkbox>` elements. Other controls, such as `<colordropdown>`, `<comboxbox>`, `<listdropdown>` and `<textbox>` elements can be copied from those delivered with Arbortext Editor, but they cannot be customized. Refer to the following file for examples of the toolbars used by Arbortext Editor.
Arbortext-path`\lib\dialogs\editwindow.xml`

Note

Do not customize the files that ship with Arbortext Editor. Use these files as templates for site-specific files you store in your `\custom` directory.

Toolbar buttons invoke ACL functions using their *command* attributes. AOM calls and events are not supported.

You can display a XUI toolbar with the ACL function `window_load_component_file()` or with the AOM `loadComponentFile` method of the **Window** class.

Use the following ACL functions to control toolbars. (*toolbar_id* is the value of the *id* attribute of the `<toolbar>` element in the XUI file.)

- Hiding and showing toolbars
  ```
  dlgitem_hide(win, toolbar_id)
  dlgitem_show(win, toolbar_id)
  ```
- Hiding and showing a toolbar while also removing and adding the toolbar name to the **View ▸ Toolbars** menu.
  ```
  dlgitem_withdraw(win, toolbar_id)
  dlgitem_display(win, toolbar_id)
  ```
- Removing toolbars
  ```
  dlgitem_remove_toolbar(win, toolbar_id)
  ```
- Temporarily removing or replacing a control in a toolbar. *control_id* is the *id* attribute of the element representing the control in the toolbar.
  ```
  dlgitem_withdraw(win, control_id)
  dlgitem_display(win, control_id)
  ```

  For example, the following function removes the `Toolbar_InsertMarkup` toolbar button in the Arbortext Editor toolbar `toolbar2`:
  ```
  dlgitem_withdraw(win, 'Toolbar_InsertMarkup');
  ```

Note

When customizing toolbars using these functions, note that Arbortext Editor distinguishes between toolbar items (such as buttons and separators) by processing the value of each one's *command*= attribute. Objects that have the same value for the attribute are indistinguishable for purposes of enabling, disabling, hiding, unhiding, and other kinds of manipulation, even if they have a different configuration. To ensure that objects are processed differently based on their configuration, for example to hide one separator and display the others, each object must have their *command*= attribute set to a different value.

For example:

```
<separator id="Separator-sep1" command="response('a')'"/
> ... <separator id="Separator-OptionalToolBtnSeparator"
withdraw="true" command="response{'b'}"/V
```

Working with Tables

The `<tablecontrol>` element displays content in rows and columns. The child element `<header>` defines the columns in the table. The child element `<row>` defines the table's rows. The `<tablecontrol>` *sortedcolumn* attribute specifies the column to use when sorting rows. (When rows are resorted in a

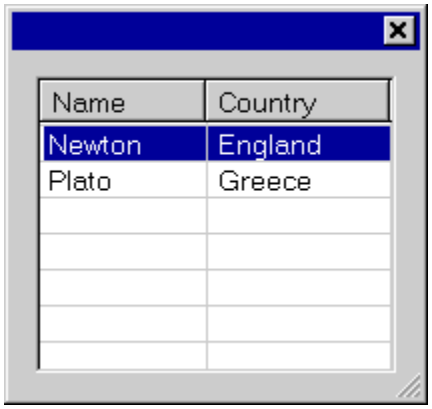
dialog box, the order of the rows in the XUI XML document is not reordered.) The<tablecontrol> *gridlines* attribute specifies that rules are displayed between rows and columns.

If the <tablecontrol> *showimages* attribute is set to true, images specified in the <row> elements will appear in the table to the left of each row.

Example

Table control

The following XUI markup defines this table:




```
<window width="125" height="200" focus="tablecontrol">
 <tablecontrol id="tablecontrol" columns="2" value="Newton" gridlines="true">
 <header>
 <column label="Name"/>
 <column label="Country"/>
 </header>
 <row>
 <cell>Newton</cell>
 <cell>England</cell>
 </row>
 <row>
 <cell>Plato</cell>
 <cell>Greece</cell>
 </row>
 </tablecontrol>
</window>
```

Working with Trees

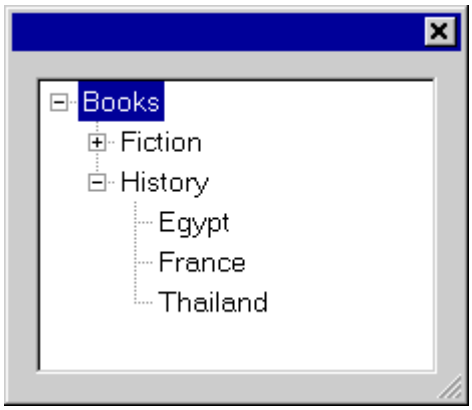
The <treecontrol> element creates a hierarchical outline view of data displayed as branches off of nodes. Each node in a <treecontrol> structure is created with a <treenode> element. A <treecontrol> element can have child <treecontrol> elements.

The <treecontrol> *branchimage*, *extraimage*, *leafimage*, *openbranchimage*, and *selectedimage* parameters let you specify images to appear to the left of node labels. The same parameters on <treenode> elements override those set on the <treecontrol> element.

Example

Tree control

The following XUI markup defines this tree control:




```
<window width="125" height="200" focus="treecontrol">
 <treecontrol id="treecontrol" height="100">
 <treenode label="Books" selected="true">
 <treenode label="Fiction">
 <treenode label="Fantasy"></treenode>
 <treenode label="Mystery"></treenode>
 <treenode label="Science"></treenode>
 </treenode>
 <treenode label="History" expanded="true">
 <treenode label="Egypt"></treenode>
 <treenode label="France"></treenode>
 <treenode label="Thailand"></treenode>
 </treenode>
 </treenode>
 </treecontrol>
</window>
```

Selecting Objects in Tree Controls

A XUI tree control has two different selection types or modes: single selection and multiple selection. The tree control selection mode is set with the *seltype* attribute on the `treecontrol` element. Legal values are `single` (the default) and `multiple`. The selection mode cannot be changed at run time.

In single selection mode, the tree control allows only one tree node to be selected at a time.

Multiple selection allows more than one tree node in the tree control to be selected at a time. Multiple selection mode is accessible using ACL only. XUI has no default rules as to how parent and child nodes are handled when selected in multiple selection mode. Such handling is the responsibility of the custom ACL application working with the XUI controls.

Detect selection changes by using the `dlgitem_add_callback` function to add a callback to the tree control and the check for `ITEM_CHANGED` events in the callback function. Use the following functions to work with tree control selections:

- `dlgitem_get_select_array(window, dlgitem, array)`

  If the tree control is in single selection mode, the array will be populated with one entry — the list tag of the only selected tree node. If the tree control is in multiple selection mode, the array will be populated with the list tags of all selected nodes in the tree control.

- `dlgitem_get_selected_appdata(window, dlgitem)`

  This function returns the application-specific data for the selected node when in single selection mode. When the tree control is in multiple selection mode, the function returns the application-specific data for the first node in the selection. To get the application-specific data of all selected items when in multiple selection mode, first find all of the selected nodes using `dlgitem_get_selected_listtag_array()`, then iterate over the returned list tags calling `dlgitem_get_appdata_at()` for each.

- `dlgitem_get_selected_listtag(window, dlgitem)`

  This function returns the list tag of the selected node when the tree control is in single selection mode. When the tree control is in multiple selection mode, it returns the list tag of the first selected tree node.

- `dlgitem_select_list_at(window, listtag, row)`

  This function selects the tree node at the position specified. When the tree control is in single selection mode, the specified tree node will be set as the only selected node. When the tree control is in multiple selection mode, the specified tree node is added to the current selection; no nodes are unselected as a result of this call when a valid row is specified. To select a single node when the tree control is in multiple selection mode, first clear any selection, then use this function to select the desired node.

- `dlgitem_get_selected_listtag_array(window, dlgitem, array)`

This function returns an array containing the list tags of all selected nodes in the tree control. When the tree control is in single selection mode the array contains a single value — the list tag of the only selected node in the tree control. When the tree control is in multiple selection mode, the array contains the list tags of each selected node.

Dragging and Dropping Tree Control Content

You can perform standard Windows drag and drop operations on XUI tree control items. XUI tree control drag and drop operations are accessible using ACL only. Drag and drop operations let you transfer data to and from the XUI tree control from and to other applications that also support drag and drop operations.

Two types of data can be transferred using a drag-and-drop operations:

- Text data — A block of text.
- File data — One or more file path names.

When a drag and drop operation begins in a XUI tree control, an ACL application is responsible for determining the type of data format that will be used, as well as providing the actual data. When a XUI tree control is the target of a drag and drop operation, the ACL application is responsible for examining the data available and determining how it is to be processed once it is dropped on the tree control. The meaning and the handling of the data in a drag and drop operation is completely up to the custom ACL application.

Drag and drop functionality is enabled by handling the following drag and drop events in your custom ACL application:

- `DRAGDROPBEGIN` — Signals the beginning of a drag and drop operation in a XUI tree control. This event is raised when the left mouse button is clicked and held down over a selected item in the source tree control and the mouse is moved. A handler for this event must be provided to allow a XUI tree control to be a drag-and-drop source. (That is, to have drag and drop operations begin in the tree control.)

  `DRAGDROPBEGIN` has the following event handler:

  `ondragdropbegin(`*`windowdlgitemddArray[]`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.
  - *ddArray[]* — The array to be filled with data. If the data format is `DDF_TEXT`, only the first element of the array contains data. All other entries in the array will be ignored for text data. If the data format is `DDF_FILE`, each element of the array contains an appropriate file path.

Return values:

- `-1` — The drag and drop operation failed to start.
- `0` — The drag and drop operation started successfully with a data format of `DDF_TEXT`.
- `1` — The drag and drop operation started successfully with a data format of `DDF_FILE`.

- `DRAGDROPENTER` — Signals a drag and drop operation has entered the target tree control. This event is raised when the mouse is first moved into the tree control during a drag and drop operation. The handler function for this event determines what the drop effect would be if a drop were attempted at the current position.

  `DRAGDROPENTER` has the following event handler:

  `ondragdropenter(`*`windowdlgitemddFmtddArray[]keyMod slisttag`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.
  - *ddFmt* — The format of the drag and drop data. A value of `0` means DDF_TEXT data. A value of `1` means DDF_FILE data.
  - *ddarray[]* — The array containing data. If the data format is `DDF_TEXT`, only the first element of the array contains data. All other entries in the array will be ignored for text data. If the data format is `DDF_FILE`, each element of the array contains an appropriate file path.
  - *keyMods* — A bitmask specifying what relevant keys are being pressed during the drag and drop operation. This mask can be a combination of the following values:
    - `0` — No keys are pressed.
    - `1` — **CTRL** key is pressed.
    - `2` — **SHIFT** key is pressed.
    - `4` — **ALT** key is pressed.
  - *listtag* — The tree control item to which the drop location is relative. The ACL application controls whether data is inserted before, after, or into this item. *listtag* may be empty if the drop position is not over a tree item.

  Return values:

  - `0` — A drop is not allowed at the current location.

- `1` — A drop of the data at the current location would result in data being copied.
- `2` — A drop of the data at the current location would result in data being moved.

The mouse cursor is set to standard Windows drag and drop cursors depending on the value returned.

- `DRAGDROPOVER` — Signals a drag and drop operation is over the target tree control. This event is raised when the mouse is over the tree control during a drag and drop operation. This handler function determines what the drop effect would be if a drop were attempted at the current position.

  `DRAGDROPOVER` has the following event handler:

  `ondragdropover(`*`windowdlgitemddFmtddArray[]keyMod slisttag`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.
  - *ddFmt* — The format of the drag and drop data. A value of `0` means DDF_TEXT data. A value of `1` means DDF_FILE data.
  - *ddarray[]* — The array containing data. If the data format is `DDF_TEXT`, only the first element of the array contains data. All other entries in the array will be ignored for text data. If the data format is `DDF_FILE`, each element of the array contains an appropriate file path.
  - *keyMods* — A bitmask specifying what relevant keys are being pressed during the drag and drop operation. This mask can be a combination of the following values:
    - `0` — No keys are pressed.
    - `1` — **CTRL** key is pressed.
    - `2` — **SHIFT** key is pressed.
    - `4` — **ALT** key is pressed.
  - *listtag* — The tree control item to which the drop location is relative. The ACL application controls whether data is inserted before, after, or into this item. *listtag* may be empty if the drop position is not over a tree item.

  Return values:

  - `0` — A drop is not allowed at the current location.
  - `1` — A drop of the data at the current location would result in data being copied.

- `2` — A drop of the data at the current location would result in data being moved.

The mouse cursor is set to standard Windows drag and drop cursors depending on the value returned.

- `DRAGDROPLEAVE` — Signals a drag-and-drop operation has left the target tree control. This event is raised when the mouse cursor leaves the tree control. Handle this event when you want to provide custom behavior when the drag and drop operation leaves a tree control.

  `DRAGDROPLEAVE` has the following event handler:

  `ondragdropleave(`*`windowdlgitem`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.

  `ondragdropenter` does not return a value.

- `DRAGDROPDROP` — Signals a drop has occurred in the target tree control. This event is raised when a drop operation is to occur. Handle this event to insert the dropped data.

  `DRAGDROPENTER` has the following event handler:

  `ondragdropenter(`*`windowdlgitemddFmtddArray[]keyMod slisttag`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.
  - *ddFmt* — The format of the drag and drop data. A value of `0` means DDF_TEXT data. A value of `1` means DDF_FILE data.
  - *ddarray[]* — The array containing data. If the data format is `DDF_TEXT`, only the first element of the array contains data. All other entries in the array will be ignored for text data. If the data format is `DDF_FILE`, each element of the array contains an appropriate file path.
  - *keyMods* — A bitmask specifying what relevant keys are being pressed during the drag and drop operation. This mask can be a combination of the following values:
    - `0` — No keys are pressed.
    - `1` — **CTRL** key is pressed.
    - `2` — **SHIFT** key is pressed.

- `4` — **ALT** key is pressed.

- *listtag* — The tree control item to which the drop location is relative. The ACL application controls whether data is inserted before, after, or into this item. *listtag* may be empty if the drop position is not over a tree item.

Return values:

- `0` — A drop is not allowed at the current location.
- `1` — A drop of the data at the current location would result in data being copied.
- `2` — A drop of the data at the current location would result in data being moved.

The mouse cursor is set to standard Windows drag and drop cursors depending on the value returned.

- `DRAGDROPEND` — Signals a drag and drop operation that started in the tree control has ended. This event is raised after the `DRAGDROPDROP` event only when the drag and drop operation started in this tree control. The event handler for this event should modify the tree control based on the resulting drop effect of the drag and drop operation. For example, if the drop effect result of the drag and drop operation is `2`, meaning the drop succeeded as a move operation, the drop target will have inserted the data. This handler should delete the source data to complete the move operation.

  `DRAGDROPEND` has the following event handler:

  `ontreectrldragdropend(`*`windowdlgitemdropEff`*`)`

  Parameters:

  - *window* — The identifier of the window containing the tree control.
  - *dlgitem* — The tree control item for which the event is occurring.
  - *dropEff* — The resulting drop effect of the drag and drop operation:
    - `0` — The drop was not allowed or was cancelled.
    - `1` — The drop succeeded as a copy operation.
    - `2` — The drop succeeded as a move operation.

  `ontreectrldragdropend` does not return a value.

Example

Drag and drop example

A sample application demonstrating how to use the XUI tree control drag-and-drop and multiple selection functionality is included in the following directory.
`Arbortext-path``\samples\XUI\`

The sample consists of the following files:

- `treectrldddlg.xml` — A XUI file defining a dockable window containing a tree control. The tree control is populated with some sample items.
- `spaceimglist7.bmp` — A bitmap graphic containing several images used for tree control items.
- `treectrlddtest.acl` — An ACL file which loads the dockable window described in `treectrldddlg.xml` and handles drag-and-drop functionality for it.

Use the following steps to run this sample application:

1. Copy `treectrldddlg.xml` and `spaceimglist7.bmp` to your `\custom\dialogs` directory.
2. Copy `treectrlddtest.acl` to your `\custom\scripts` directory.
3. Start Arbortext Editor.
4. At the Arbortext Editor command line, enter:
   ```
   source treectrlddtest.acl
   ```
   A window containing a tree control docked to the left side of the Arbortext Editor window will be displayed.

This sample can demonstrate the transfer of both text and file data to and from the tree control by dragging and dropping the data. For drag-and-drop operations that originate in the tree control, the convention used in this sample is:

- If a tree item has application-specific data associated with it, the application-specific data is a file path and the tree item should be dragged and dropped as file data.
- If a tree item has no application-specific data associated with it, the tree item should be dragged and dropped as text data with the item label as the data.

Initially, the tree control contains a set of sample items that have no application-specific data.

For multiple selection, no dragging and dropping is allowed if a parent-child combination is selected. This sample application provides no rules for how this should be handled, so it is not allowed.

- Demonstrating dragging and dropping text data within the tree control:
  1. Select an item in the tree control.
  2. Click the left mouse button on the selected item and drag it over another item in the tree control.
  3. Release the left mouse button to move the item. over an item in the tree control. (Pressing **CTRL** while dragging and dropping copies the item.)

When a drag and drop operation beginning in the tree control results in a move operation, the original item will be deleted from the tree control automatically. The ACL application does not need to handle removing the item.

- Demonstrating dragging and dropping text data to Arbortext Editor:

  1. Select an item in the tree control.
  2. Click the left mouse button on the selected item and drag it over the Arbortext Editor Edit window.
  3. Release the left mouse button over the Arbortext Editor window to drop the data into Arbortext Editor. The item label text will be copied into the Arbortext Editor window.

     Pressing **CTRL** during the drag and drop operation has no effect. For dragging and dropping text data, Arbortext Editor always performs a copy operation.

- Demonstrating dragging and dropping text data from an external application:

  1. Start WordPad.
  2. Open a document in WordPad or type some text in an empty document.
  3. Select some text in the WordPad document and drag and drop it over the tree control. The text will be inserted as an item in the tree control at the drop location and removed from the original document. Pressing **CTRL** while dragging and dropping copies the text to the tree control while leaving it in the original document.

     Arbortext Editor does not support dragging and dropping text from an Edit window to a tree control. Text can only be dragged and dropped between Edit windows.

- Demonstrating dragging and dropping file data from an external application:

  1. Open Windows Explorer and navigate to a directory containing some files. Make sure the files do not contain critical data and they are backed up.
  2. Select one or more files in Windows Explorer and drag and drop them to the tree control. Each file will be inserted as an item in the tree control. The file name will be used as the item label and the application-specific data of the item will be set to the file path. The original files will also remain in their original locations.

- Demonstrating dragging and dropping file data to an external application:
  1. Select a file item in the tree control placed there in the previous demonstration.
  2. Drag and drop the file to a different directory in Windows Explorer. The file will move from the original location to the new location. (The original tree control item will remain, however.) Pressing **CTRL** during the drag and drop operation will place a copy of the file in the new location.
- Demonstrating dragging and dropping file data to Arbortext Editor:
  1. Select a file item in the tree control placed there in the first file demonstration.
  2. Drag the tree control item over Arbortext Editor and drop it. The file will open in Arbortext Editor.

Working with Dockable Dialog Boxes

Dockable dialog boxes can be displayed standalone or be dragged into a Arbortext Editor edit window and docked on one edge of the edit window. If you drag a docked dialog box away from the edge of the edit window, the dialog box undocks. A dockable dialog box can contain all controls that are allowed in a standard dialog box except for toolbars and menubars. (`<menubar>` and `<toolbar>` elements in the file are ignored.)

The markup for a dockable dialog box is the same as that for a standard dialog box. Dockable dialog boxes are specified using the attributes *enabledocking* and *dock* of the `<window>` element.

- *enabledocking* — Specifies the edges of the edit window the dialog box can dock to.

  The default value of *enabledocking* is `none`. If *enabledocking* is `none`, the XUI file will be displayed as a non-dockable (standard) dialog box.
- *dock* — Specifies the docking state of the dockable dialog box.

  The default value of *dock* is `none`. The value of *dock* must be one of the locations specified by *enabledocking*. Otherwise, *dock* will be ignored.

The following example shows a valid pairing of attributes:

```
<window enabledocking="leftright" dock="left" >
...
</window>
```

Whether a dialog box is dockable or not is determined at the dialog box creation time. After a dialog box is created, the dialog box cannot change between dockable and undockable even if *enabledocking* is modified or the AOM `enableDocking` method is called.

The *modal* attribute in the `<window>` element takes precedence over the *enabledocking* attribute. If *modal* has a value of `true`, the XUI file will be displayed as a standard modal dialog box regardless of the value of *enabledocking*.

Geometry

When a dockable dialog box is docked in an edit window, users still can resize the dialog box by dragging the splitter separating the docked dialog box and the edit window.

Arbortext Editor remembers three sizes for each dockable dialog type:

- the size when the dialog box is docked horizontally
- the size when the dialog box is docked vertically
- the size when the dialog box is floating

Therefore, when a dockable dialog box is displayed or docked, Arbortext Editor will set the dialog box size to the size of the previous dialog box of the same type.

Example

Refer to the dockable dialog box example contained in the file:
Arbortext-path`\samples\XUI\dockableDialog.xml`

Dockable Dialog Boxes and the AOM

The methods to create and display a dockable dialog box are the same as those for a standard dialog box. For example:

```
var dialog = Application.createDialogFromFile('c:\temp\myxuifile');
dialog.dock = 0; // override the dock attribute in the XML file
dialog.show();
```

Identifying the Parent Window of a Dialog Box

The methods `Application.createDialogFromFile()` and `Application.createDialogFromDocument()` each take a parameter, *parent*, for specifying the parent window of the dialog box. If the parent is not specified, the default parent is the current active window.

The *parent* parameter is useful when a user creates a dockable dialog box in a document type startup file or a document startup file. At the time these startup files are executed, the active document is the document to be brought up, and the edit window for the document is not yet displayed.

Users can get the new edit window from the active document and assign the correct parent to the new dockable dialog box as follows:

```
var parent = Application.activeDocument.defaultView.window;
var window = Application.createDialogFromFile('my.xml',null,parent);
window.dock = window.DOCK_LEFT;
window.show();
```

Embedding XUI Dialog Box Controls in a Document

You can embed XUI dialog box controls in the content of a document displayed in Arbortext Editor. Users can use these controls to interact with other data sources, take advantage of ActiveX controls, select values from restricted lists, and so on.

Use the following steps to embed XUI dialog box controls in a document. (Examples follow.)

1. Create a XUI file defining the dialog box controls you want embedded in the document. Use events in `<script>` elements to define the actions to occur when the events are triggered.
2. Determine which element in your DTD you want to replace with the contents of the XUI file defining the dialog box controls.
3. Add a `<XuiControl>` element to the `<Specials>` section of the DTD's `.dcf` file specifying the element in the DTD you want to replace with the contents of the XUI file.

   The `<XuiControl>` element has the following attributes:

   - `element` — The name of the element to be associated with an embedded XUI control.
   - `xuiFileName` — The file that describes the XUI dialog box controls.
   - `condition` — An XPath expression specifying whether to create the dialog box controls based on one or more attributes or values. If the expression evaluates to `TRUE`, the controls will be created or attached to. If the expression evaluates to `FALSE`, the controls will not be used.
4. Create a new document based on the DTD and insert the chosen element.

   The XUI controls appear in the Edit pane embedded in the flow of the document. The Document Map shows the element with an icon signifying it as embedded XUI markup. Hover the mouse pointer over the icon to display the name of the XUI file referenced by the element.

By default, the XUI controls are displayed embedded in the document. You can display the XUI markup inline instead of the rendered controls using the ACL `set` command `set dialogdisplay=off`. `set dialogdisplay=on` will again display the embedded controls.

When using embedded XUI dialog box controls, be aware that only one occurrence of a specific element's dialog box can be displayed at one time. If the same element appears in multiple open windows, only one occurrence of the element will be displayed as an embedded XUI control.

Example

Embedded combo box

This example shows how to embed a combo box in a document. The user can select an item from the combo box and have it inserted at the current cursor location.

1. Create a XUI file defining the combo box. In the following example, the script element causes the chosen value to be inserted at the current cursor location. The XUI file is saved as `units.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.1//EN"
 "xui.dtd">
<window orient="vertical"
 modal="false"
 title="UnitSelection">
<label label="Select a unit of measure:"/>
<combobox value="grams" type="dropdown">
<listitem label="grams"/>
<listitem label="kilograms"/>
<listitem label="milligrams"/>
<listitem label="ounces (Troy)"/>
<listitem label="ounces (U.S.)"/>
<listitem label="pounds (Troy)"/>
<listitem label="pounds (U.S.)"/>
<script type="application/x-javascript" ev:event="domactivate">
 var selection = Application.event.target.getAttribute("value");
 var textnode = Application.activeDocument.createTextNode(selection);
 Application.event.view.window.ownerNode.appendChild(textnode);
</script>
</combobox>
</window>
```

2. For purposes of this example, the `action` element is defined in the `.dcf` file as the element to be replaced with the dialog box generated by `units.xml`:

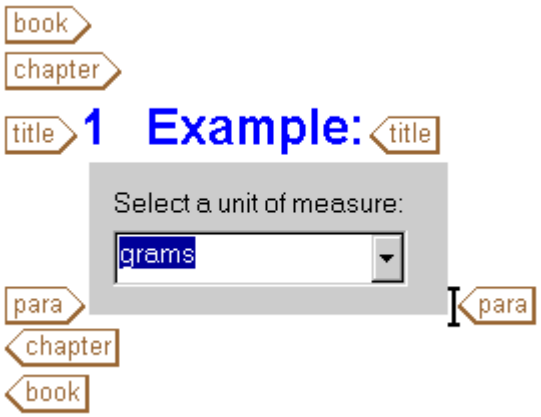
```
<Specials>
```

```
<XuiControl element="action" xuiFileName="units.xml"/>
```

3. Embed the combo box in the document by inserting the `action` element. For example, the source XML:

```
<book>
<title>Units of Measure Example</title>
<chapter>
<title>Example:</title>
<para><action></action></para>
</chapter>
</book>
```

appears in Arbortext Editor as follows:



Selecting an item from the combo box will insert the contents of the `<combobox> value` element at the current cursor location.

XUI Display Recommendations

When implementing XUI dialog boxes, you should add as many XUI controls to a dialog box as possible in a single pass to improve the display refresh of XUI dialog boxes.

For example, the following Java code adds ten controls to the dialog box one by one:

```
Element box = _xui.getElementById("AddControlsBox");
for(int i = 0; i < 10; ++i) {
 Element textbox = _xui.createElement("textbox");
 box.appendChild(textbox);
}
```

The dialog box display refresh would be improved by adding all ten controls to the dialog box at once as in the follow example:

```
DocumentFragment frag = _xui.createDocumentFragment();
for(int i = 0; i < 10; ++i) {
 Element textbox = _xui.createElement("textbox");
 frag.appendChild(textbox);
```

```
}
Element box = _xui.getElementById("AddControlsBox");
box.appendChild(frag);
```

For optimum display speed when clearing all `listitems` from a `combobox` or `listbox`, delete the items in order beginning with the **firstChild**.

XUI Element Reference

Arbortext Editor supports the following XUI elements for creating dialog box controls. The document type for XUI XML documents is *`Arbortext-path`*`\doctypes\xui\xui.dtd`.

<activex> Element

The `<activex>` element is a container for an ActiveX control. Refer to the *Working with ActiveX controls* chapter of the *Programmer's Reference* for details on using ActiveX controls. The script for an ActiveX control must be either Jscript or VBscript. It cannnot be JavaScript.

The element can have the following child elements:

`<script>`, `<param>`

The `<activex>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *progid* = *ID*

  Specifies the program ID of the ActiveX control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<box> Element

The `<box>` element creates a box container. The element can have the following child elements:

<activex>, <box>, <button>, <checkbox>, <colordropdown>, <combobox>, <description>, <grid>, <groupbox>, <label>, <listbox>, <morph>, <picturebox>, <radio>, <radiogroup>, <script>, <separator>, <slider>, <spacer>, <spinner>, <tabbox>, <tablecontrol>, <textbox>, <treecontrol>, <unitdimensionbox>

The <box> element has the following attributes:

- *align* = start | center | end

  Default is start. Specifies how space not used by the children is laid out.

  If *orient* is horizontal, start aligns children at the top of the container. places space below children. If *orient* is vertical, start aligns children at the left of the container. center evenly distributes space above and below children. end places space above children.

- *childrensize* = vary | equalwidth | equalheight | equal

  Specifies whether the children of this control will have the same height and width as each other. Default is vary. If vary, the height and width of the children will not be restricted. If equalwidth, the width of the children will be the same as that of the widest child. If equalheight, the height of the children will be the same as that of the tallest child. If equal, the width of the children will be the same as that of the widest child and the height of the children will be the same as that of the tallest child..

- *disabled* = true | false

  Default is false. If true, the control ignores all user interface events directed toward the control.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = true | false

  Default is false. If true, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *orient* = vertical | horizontal

  Default is horizontal. Specifies the layout for the container children.

- *pack* = start | center | end | spread | stretch

  Default is spread. Specifies how space not used by the children is laid out. start places space after children. center evenly distributes space before and after children. end places space before children. spread evenly

distributes space before, between, and after children. `stretch` evenly distributes space between children, with no space before or after children.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<button> Element

The `<button>` element creates a button control. The element can have the following child element:

`<script>`

The `<button>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *command* = *CDATA*

  Specifies the Arbortext ACL command to execute. (Toolbars only.)

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *dropdown* = *CDATA*

  The id or name of a dropdown menu for this control that is displayed when the button is activated. The value can be specified in an *id* attribute of a `<popupmenu>` element, or it can be the name of a shortcut menu loaded from a menu configuration file.

Adding a drop down menu to a button visually changes the button. A down arrow is displayed in the button to the right of the label.

If this attribute has no value, or if the value is not the id of a menu defined in the XUI dialog box file or that of a name loaded from a menu configuration file, no menu is displayed and the button performs its default activation behavior.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *label* = *CDATA*

  Specifies the text to display within the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control is not resizeable.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.
- *type* = `accept` | `cancel` | `help`

  Specifies a key event that activates the button when the event occurs within the dialog box. `accept` is typically activated by the **Enter** key being pressed. `cancel` is typically activated by the **ESC** key being pressed.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<cell> Element

The `<cell>` element creates a cell within a table. The element can have no child elements.

The `<cell>` element has the following attributes:

- *id* = *ID*

  Identifies the control.
- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in the cell. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.
- *selected* = `true` | `false`

  Default is `false`. If the parent `tablecontrol` element's type attribute is set to `cell`, clicking in the table selects the cell and sets the *selected* attribute to `true`.

<checkbox> Element

The `<checkbox>` element creates a check box control. The element can have the following child element:

`<script>`

The `<checkbox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *checked* = `true` | `false`

  Default is `false`. If `true`, the control has a checked state.

- *checkstate* = `checked` | `unchecked` | `indeterminate`

  Default is `unchecked`. If `checked`, the item is selected. If `unchecked`, the item is not selected. If `indeterminate`, the item displays no state. (It is disabled.)

- *command* = *CDATA*

  Specifies the Arbortext ACL command to execute. (Checkbox toolbar button only.)

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *label* = *CDATA*

  Specifies the text to display next to the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control is not resizeable.

- *statustext* = *CDATA*

  Specifies the text to display in status controls.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *type* = `twostate` | `threestate`

  Default is `twostate`. If `twostate`, the control can be either checked or unchecked (specified by *checked*). If `threestate`, the control can be checked, unchecked, or indeterminate (specified by *checkstate*).

- *width* = *CDATA*

Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<colordropdown> Element

The `<colordropdown>` element displays a color selection control from which the user picks a color. *value* is set to the value of the selected color.

The `<colordropdown>` element can have the following child element:

`<script>`

The `<colordropdown>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *command* = *CDATA*

  Specifies the Arbortext ACL command to execute. (Colordropdown toolbar buttons only.)

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *palette* = `foreground` | `background`

  Default is `foreground`. If `foreground`, the foreground color palette is displayed. If `background`, the background color palette is displayed.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control is not resizeable.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *value* = *CDATA*

  Specifies the selected color. Values are the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *valuetype* = `name` | `spec`

  Default is `name`. If set to a value of `name`, when a color is selected in the color dropdown control and a standard HTML color name exists for the color, the *value* attribute of the `<colordropdown>` element is set to the name of the color. When no HTML color name exists for the selected color, the *value* attribute of the `<colordropdown>` element is set to the hexidecimal color code specifying the red, green, and blue intensity values defining the color.

  If *valuetype* is set to a value of `spec`, the *value* attribute of the `<colordropdown>` element will is always set to the hexidecimal color code defining the color selected in the color dropdown control.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<column> Element

The `<column>` element creates a column in a table. The element can have no child elements.

The `<column>` element has the following attributes:

- *align* = `start` | `center` | `end`

  Default is `start`. Specifies how space not used by the children is laid out. `start` places space below children. `center` evenly distributes space above and below children. `end` places space above children.
- *columnresize* = `true` | `false`

  Default is `true`. If `true`, the column resizes to the maximum available width in the table control. If `false`, the column size is static.
- *columnwidth* = `CDATA`

  Number specifying the width in pixels to allocate for the column.
- *id* = *ID*

  Identifies the control.
- *label* = *CDATA*

  Specifies the text to display next to the control.

<combobox> Element

The `<combobox>` element creates a simple, dropdown, or read-only dropdown box control. The element can have the following child elements:

`<listitem>`, `<script>`

The `<combobox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction.

  If `natural`, the control allows horizontal resizing. The control also allows vertical resizing only if *type* = `simple`.
- *sorted* = `true` | `false`

  Default is `false`. Specifies whether list items should be alpha-numerically sorted.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.
- *type* = `dropdown` | `dropdownlist` | `simple`

  Default is `simple`. If `simple`, the combobox is a standard combobox. If `dropdown`, it is a drop-down combobox. If `dropdownlist`, it is a read-only drop-down combobox.
- *value* = *CDATA*

  Specifies the current value for the list.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<contextmenu> Element

The `<contextmenu>` element creates a menu control containing context-sensitive menu items. The element can have the following child elements:

`<menutitem>`, `<script>`

The `<contextmenu>` element has the following attributes:

- *id* = *ID*

  Identifies the control.
- *name* = *CDATA*

  .

  Can you give me a description of what name defines?
- *showimages* = `true` | `false`

  Default is `false`. If `true`, images are displayed to the left of each `menuitem` with an image defined.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<contextmenugroup> Element

The `<contextmenugroup>` element is the parent element for `<contextmenu>` controls. The element can have the following child elements:

`<contextmenu>`

The `<contextmenugroup>` element has the following attributes:

- *id* = *ID*

  Identifies the control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<datetime> Element

The `<datetime>` element used to select a date using the Windows month calendar control. `<datetime>` displays the selected date and a down arrow. Clicking on the down arrow displays the calender from which the user can select a particular date.

The `<datetime>` element has the following attributes:

- *disabled* = `true` | `false`

Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *ev:defaultAction* = `cancel` | `perform`

  Specifies if, after processing of all listeners for the event at the current element, the default action for the event (if any) should be performed or not. If `cancel`, the default action is cancelled (if the event type can be cancelled). If `perform`, the default action is performed.

- *ev:event* = *CDATA*

  The event type for which the listener is being registered. The value must be an XML Name. For information on working with events, refer to Specifying Event Listeners on page 121.

- *ev:handler* = *CDATA*

  Specifies the URI of an element that defines the action that should be performed if the event reaches the observer. If *ev:handler* is not supplied, the handler is the element that the event attribute is on.

- *ev:observer* = *ID*

  Identifies the element with which the event listener is to be registered. If *ev:observer* is not supplied, the observer is the element that the event attribute is on.

- *ev:phrase* = `capture` | `default`

  Specifies when the listener will be activated by the desired event. If `default`, the listener is activated during bubbling or target phase. If `capture` the listener is activated during the capturing phase.

- *ev:propagate* = `stop` | `continue`

  Specifies whether after processing all listeners at the current node, the event is allowed to continue on its path (either in the capture or the bubble phase). If `stop`, event propogation stops. If `continue`, event propagation continues.

- *ev:target* = *ID*

  Identifies the target element of the event (that is, the node that caused the event). If *ev:target* is supplied, only events that match both the event and target attributes will be processed by the associated event handler.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal resizing.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *value* = *CDATA*

  Specifies the default value of the control. This value is the number of seconds since the epoch (00:00:00 UTC on January 1, 1970). When a user selects a date from the calendar, the value is set to the first second of that day (for the specific time zone and any daylight savings time in effect of the workstation being used).

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<description> Element

The `<description>` element creates a description control containing text. The element can take no child elements. The `<description>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.
- *fontsize* = *CDATA*

  Specifies the control's font size in points.
- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.
- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.
- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.
- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *multiline* = `true` | `false`

  Default is `false`. If `true`, the edit field may wrap.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal resizing, but not vertical resizing.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *strikeout* = `true` | `false`

Default is `false`. If `true`, the text contains strike outs.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *underline* = `true` | `false`

  Default is `false`. If `true`, the text is underlined.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

- *xml:space* = *FIXED*

  Default is `preserve`. When `preserve` is enabled, spaces and linebreak characters in the element's content are preserved when the XUI file is processed by methods such as `Application.createDialogFromFile()` and `Application.createDialogFromDocument()`.

<grid> Element

The `<grid>` element creates a grid container. The element can have the following child elements:

`<box>`, `<button>`, `<checkbox>`, `<colordropdown<>`, `<combobox>`, `<description>`, `<grid>`, `<groupbox>`, `<label>`, `<listbox>`, `<morph>`, `<picturebox>`, `<radio>`, `<radiogroup>`, `<script>`, `<separator>`, `<slider>`, `<spacer>`, `<spinner>`, `<tabbox>`, `<tablecontrol>`, `<textbox>`, `<treecontrol>`, `<unitdimensionbox>`

The `<grid>` element has the following attributes:

- *cellalign* = `bottomcenter` | `bottomleft` | `bottomright` | `middlecenter` | `middleleft` | `middleright` | `topcenter` | `topleft` | `topright`

  Default is `middleleft`. Specifies how the cell children are aligned.

- *columns* = *CDATA*

  Specifies the number of columns.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<groupbox> Element

The `<groupbox>` element creates a group box control. The element can have the following child elements:

`<box>`, `<button>`, `<checkbox>`, `<colordropdown>`, `<combobox>`, `<description>`, `<grid>`, `<groupbox>`, `<label>`, `<listbox>`, `<morph>`, `<picturebox>`, `<radio>`, `<radiogroup>`, `<script>`, `<separator>`, `<slider>`, `<spacer>`, `<spinner>`, `<tabbox>`, `<tablecontrol>`, `<textbox>`, `<treecontrol>`, `<unitdimensionbox>`

The `<groupbox>` element has the following attributes:

- *align* = `start` | `center` | `end`

  Default is `start`. Specifies how space not used by the children is laid out. `start` places space below children. `center` evenly distributes space above and below children. `end` places space above children.

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`,

`gray3`, `gray4`, `gray5`. Colors can also be a string of `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *childrensize* = `vary` | `equalwidth` | `equalheight` | `equal`

  Specifies whether the children of this control will have the same height and width as each other. Default is `vary`. If `vary`, the height and width of the children will not be restricted. If `equalwidth`, the width of the children will be the same as that of the widest child. If `equalheight`, the height of the children will be the same as that of the tallest child. If `equal`, the width of the children will be the same as that of the widest child and the height of the children will be the same as that of the tallest child..

- *clip* = `true` | `false`

  Default is `false`. If `true`, controls that do not fit within the groupbox control are truncated. If `false`, the controls are not truncated.

  The value of *clip* is ignored if *scroll* is set to `none`.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *label* = *CDATA*

  Specifies the text that appears in the border.

- *orient* = `vertical` | `horizontal`

  Default is `horizontal`. Specifies the layout for the container children.

- *pack* = `start` | `center` | `end` | `spread` | `stretch`

Default is `spread`. Specifies how space not used by the children is laid out. `start` places space after children. `center` evenly distributes space before and after children. `end` places space before children. `spread` evenly distributes space before, between, and after children. `stretch` evenly distributes space between children, with no space before or after children.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *scroll* = `none` | `both` | `vertical` | `horizontal`

  Default is `none`. Specifies which, if any, scroll bars are displayed in the control. `none` specifies that no scroll bars are displayed. `vertical` specifies that only a vertical scroll bar is displayed. `horizontal` specifies that only a horizontal scroll bar is displayed. `both` specifies that vertical and horizontal scroll bars are displayed.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<header> Element

The `<header>` element is the parent element for defining columns in a table. The element can have the following child elements:

`<column>`

The `<header>` element has the following attribute:

- *id* = *ID*

  Identifies the control.

<image> Element

The `<image>` element specifies an id for and location of a graphical image. The element can have no child elements:

The `<image>` element has the following attributes:

- *id* = *ID*

Specifies an id for the image.

- *path = CDATA*

  Specifies the path and file name of the image. *path* can also be one of the MFC predefined image names: `#emergency`, `#information`, `#question`, and `#warning`.

  If the path is not an absolute path (or a path relative to the current directory), Arbortext Editor will first search the directories specified with `set dialogspath`. If the file is not located in those directories, Arbortext Editor will search the directories specified in the list of graphics paths.

- *transparentcolor = CDATA*

  Specifies the color in the source image to treat as the transparent color. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a # followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

<imagegroup> Element

The `<imagegroup>` element is the parent element for defining images. The element can have the following child elements:

`<image>`, `<imagelist>`

The `<imagegroup>` element has the following attribute:

- *id = ID*

  Specifies an id for the control.

<imagelist> Element

The `<imagelist>` element specifies a graphic file containing multiple images of identical widths and heights. For example, if *path* specifies a graphic file with a width of 288 and height of 16, and *imagewidth* is set to a value of 16, `imagelist` will contain 18 `<image>` elements, each defining an image 16 pixels wide by 16 pixels high.

The element can have the following child element:

`<image>`

The `<imagelist>` element has the following attributes:

- *id = ID*

  Specifies an id for the control.

- *imagewidth = CDATA*

Specifies the width in pixels of the images in the graphic file defined by *path*.

- *path* = *CDATA*

  Specifies the path and file name of the graphic file containing multiple images.

- *transparentcolor* = *CDATA*

  Specifies the color in the source image to treat as the transparent color for each image. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

<label> Element

The `<label>` element creates a label control. The element can have no child elements. The `<label>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *control* = *IDREF*

  When this control receives focus, focus should transfer to the dialog box control with this specified ID.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.
- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *label* = *CDATA*

  Specifies the text to display within the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control is not resizeable.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<listbox> Element

The `<listbox>` element creates a list box control. The element can have the following child elements:

`<listitem>`, `<script>`

The `<listbox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows resizing in both directions.
- *sorted* = `true` | `false`

  Default is `false`. Specifies whether list items should be sorted.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.
- *type* = `single` | `multiple`

  Default is `single`. If `single`, only one item can be selected in the list at one time. If `multiple`, more than one item can be selected in the list.
- *value* = *CDATA*

  Specifies the current value for the list.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<listdropdown> Element

The `<listdropdown>` element creates a toolbar dropdown list control. The element can have the following child elements:

`<listitem>`, `<script>`

The `<listdropdown>` element has the following attributes:

- *backgroundcolor* = *CDATA*

Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *command* = *CDATA*

  Specifies the Arbortext ACL command to execute.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows resizing in both directions.

- *sorted* = `true` | `false`

  Default is `false`. Specifies whether list contents should be sorted.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *title* = *CDATA*

  The text displayed in the frame title bar.

- *value* = *CDATA*

  Specifies the current value for the list.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<listitem> Element

The `<listitem>` element is a child of the `<combobox>` and `<listbox>` controls. The `<listitem>` element has the following attributes:

- *appdata* = *CDATA*

  Specifies a value for later reference.

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *label* = *CDATA*

  Specifies the text to display for the list item.

- *selected* = `true` | `false`

  Default is `false`. If `true`, the item is selected. If `false`, the item is not selected.

<menubar> Element

The `<menubar>` element is the parent element for creating a collection of menu selections. The element can have the following child elements:

`<menuitem>`, `<script>`

The `<menubar>` element has the following attributes:

- *id* = *ID*

  Identifies the control.

- *showimages* = `true` | `false`

  Default is `false`. If `true`, images are displayed to the left of each `menuitem` with an image defined.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<menugroup> Element

The `<menugroup>` element is a container for specifying the shortcut and dropdown menus available to the current dialog box. The element can have the following child elements:

`<popupmenu>`

The `<menugroup>` element has the following attributes:

- *id* = *ID*

Identifies the control.

<menuitem> Element

The `<menuitem>` element creates a list of menu selections (including submenus). The element can have the following child elements:

`<menuitem>`, `<script>`

The `<menuitem>` element has the following attributes:

- *checked* = `true` | `false`

  Default is `false`. If `true`, the control has a checked state.
- *command* = *CDATA*

  Specifies the Arbortext ACL command to execute.
- *id* = *ID*

  Identifies the control.
- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.
- *label* = *CDATA*

  Specifies the text to display for the list item.
- *shortcut* = `CDATA`

  Specifies the keyboard shortcut for selecting this control.
- *type* = `button` | `radio` | `separator` | `toggle`

  Specifies the type of menu item.

  - `button` — The menu item is displayed as a button. When selected, the button will execute the actions defined by a `script` child element.
  - `radio` — The menu item is displayed as a radio button. When selected, the button will execute the actions defined by a `script` child element.
  - `separator` — The menu item appears as a separation line between adjacent items.
  - `toggle` — When activated, the item will execute the actions defined by a `<script>` child element and display showing it has been activated. When deactivated, the item is displayed showing it is not activated.
- *withdraw* = `true` | `false`

Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<morph> Element

The `<morph>` element creates a control with a layout that dynamically rearranges its contents as the dialog box is resized. The element can have the following child elements:

`<box>`, `<button>`, `<checkbox>`, `<colordropdown>`, `<combobox>`, `<description>`, `<grid>`, `<groupbox>`, `<label>`, `<listbox>`, `<morph>`, `<picturebox>`, `<radio>`, `<radiogroup>`, `<script>`, `<separator>`, `<slider>`, `<spacer>`, `<spinner>`, `<tabbox>`, `<tablecontrol>`, `<textbox>`, `<treecontrol>`, `<unitdimensionbox>`

The `<morph>` element has the following attributes:

- *columns* = *CDATA*

  Specifies the number of columns.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *gridfitheight* = *CDATA*

  If the control height in pixels is less than or equal to *gridfitheight*, then the control has a grid layout. If the control height in pixels is greater than *gridfitheight*, then the control has a `tabbox` layout.

- *gridfitwidth* = *CDATA*

  If the control width in pixels is less than or equal to *gridfitwidth*, then the control has a grid layout. If the control width in pixels is greater than *gridfitwidth*, then the control has a `tabbox` layout.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<param> Element

The `<param>` element provides parameters for the ActiveX control defined with the parent `<activex>` element. This element can have no child elements.

The `<param>` element has the following attributes:

- *name* = *CDATA*

  The name of one of the ActiveX control's properties.

- *value* = *CDATA*

  Sets the value of the property specified by *name*.

<picturebox> Element

The `<picturebox>` element displays an image in a dialog box.

The `<picturebox>` element can have the following child element:

`<script>`

The `<picturebox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control is not resizeable.

- *statustext* = *CDATA*

Specifies the text to display in status controls.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<popupmenu> Element

The `<popupmenu>` element is a container for container for items specified within a shortcut or dropdown shortcut menu. The element can have the following child elements:

`<menuitem>`

The `<popupmenu>` element has the following attributes:

- *id* = *ID*

  Identifies the control.

<radio> Element

The `<radio>` element creates a radio button in a `<radiogroup>` control. The element can have the following child element:

`<script>`

The `<radio>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *checked* = `true` | `false`

  Default is `false`. If `true`, the control has a checked state.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.
- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.
- *fontsize* = *CDATA*

  Specifies the control's font size in points.
- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.
- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.
- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.
- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.
- *label* = *CDATA*

  Specifies the text to display next to the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows

resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<radiogroup> Element

The `<radiogroup>` element creates a radio group control. The element can have the following child elements:

`<grid>`, `<radio>`, `<script>`

The `<radiogroup>` element has the following attributes:

- *align* = `start` | `center` | `end`

  Default is `start`. Specifies how space not used by the children is laid out. `start` places space below children. `center` evenly distributes space above and below children. `end` places space above children.

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *childrensize* = `vary` | `equalwidth` | `equalheight` | `equal`

  Specifies whether the children of this control will have the same height and width as each other. Default is `vary`. If `vary`, the height and width of the children will not be restricted. If `equalwidth`, the width of the children will be the same as that of the widest child. If `equalheight`, the height of the children will be the same as that of the tallest child. If `equal`, the width of the children will be the same as that of the widest child and the height of the children will be the same as that of the tallest child..

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *label* = *CDATA*

  Specifies the text to display next to the control.

- *orient* = `vertical` | `horizontal`

  Default is `horizontal`. Specifies the layout for the container children.

- *pack* = `start` | `center` | `end` | `spread` | `stretch`

  Default is `spread`. Specifies how space not used by the children is laid out. `start` places space after children. `center` evenly distributes space before and after children. `end` places space before children. `spread` evenly distributes space before, between, and after children. `stretch` evenly distributes space between children, with no space before or after children.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<row> Element

The `<row>` element creates a row within a table. The element can have the following child elements:

`<cell>`

The `<row>` element has the following attributes:

- *id* = *ID*

  Identifies the control.

- *image* = *IDREF*

  Specifies by reference the graphical image to be displayed in this control. The value of *image* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *selected* = `true` | `false`

  Default is `false`. If `true`, the item is selected. If `false`, the item is not selected.

<script> Element

The `<script>` element specifies which script interpreter to use. The content of the element specifies the statements to be executed. The element can have no child elements.

The `<script>` element has the following attributes:

- *ev:defaultAction* = `cancel` | `perform`

  Specifies if, after processing of all listeners for the event at the current element, the default action for the event (if any) should be performed or not. If `cancel`, the default action is cancelled (if the event type can be cancelled). If `perform`, the default action is performed.

- *ev:event* = *CDATA*

  The event type for which the listener is being registered. The value must be an XML Name. For information on working with events, refer to Specifying Event Listeners on page 121.

- *ev:handler* = *CDATA*

  Specifies the URI of an element that defines the action that should be performed if the event reaches the observer. If *ev:handler* is not supplied, the handler is the element that the event attribute is on.

- *ev:observer = ID*

  Identifies the element with which the event listener is to be registered. If *ev:observer* is not supplied, the observer is the element that the event attribute is on.

- *ev:phrase* = `capture` | `default`

  Specifies when the listener will be activated by the desired event. If `default`, the listener is activated during bubbling or target phase. If `capture` the listener is activated during the capturing phase.

- *ev:propagate* = `stop` | `continue`

  Specifies whether after processing all listeners at the current node, the event is allowed to continue on its path (either in the capture or the bubble phase). If `stop`, event propogation stops. If `continue`, event propagation continues.

- *ev:target = ID*

  Identifies the target element of the event (that is, the node that caused the event). If *ev:target* is supplied, only events that match both the event and target attributes will be processed by the associated event handler.

- *id = ID*

  Identifies the control.

- *type = CDATA*

  Specifies the script interpreter to use when executing the statements. Use the following strings for specifying interpreters:

  - Rhino JavaScript — `application/x-javascript`
  - Microsoft JScript — `application/x-jscript`
  - Microsoft VBScript — `application/x-vbscript`

- *usage* = `direct` | `indirect`

  Default is `direct`. If `direct`, the observer of the event is the parent element of the script element. If `indirect`, the handler of the event is another element. That other element is identified by the *ev:handler* attribute. Its value is the ID of the element that should handle the event.

  Default is `direct`. If `direct`, the observer of the event is the parent element of the script element. If `indirect`, the observer of the event is another element. That other event has an *ev:handler* attribute which points to this script element.

- *xml:space = FIXED*

  Default is `preserve`. When `preserve` is enabled, spaces and linebreak characters in the element's content are preserved when the XUI file is

processed by methods such as `Application.createDialogFromFile()` and `Application.createDialogFromDocument()`.

<separator> Element

The `<separator>` element creates a line separating a dialog box into parts. The element creates a horizontal line when it is in a vertical box. The element creates a vertical line when it is in a horizontal box. The element can have no child elements.

The `<separator>` element has the following attributes:

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If the parent box is oriented vertically, `natural`, resizes horizontally. If the parent box is oriented horizontally, `natural`, resizes vertically.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<slider> Element

The `<slider>` element creates a control used for selecting a value in a range by moving an indicator along a horizontal bar. The element can have the following child element:

`<script>`

The `<slider>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`,

`gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *increment* = *CDATA*

  Specifies the value by which the control can be increased and decreased.

- *maximum* = *CDATA*

Specifies the lowest number in the control's range.

- *minimum* = *CDATA*

  Specifies the highest number in the control's range.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal resizing.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *ticfrequency* = *CDATA*

  Specifies the distance between tic marks on the control's horizontal scale. For example, if *increment* is set to `3` and *ticfrequency* is set to `5`, the scale will have a tic mark every 15 units. If *ticfrequency* is set to `0`, no tic marks are displayed.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *value* = *CDATA*

  Specifies the default value of the control.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<spacer> Element

The `<spacer>` element is a hidden box (container) used for placing flexible space between controls. The element can have no child elements.

The `<spacer>` element has the following attributes:

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows resizing in both directions.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<spinner> Element

The `<spinner>` element creates a control used for specifying a value by entering the value in an edit field or by changing the current value using up and down arrows. The element can have the following child element:

`<script>`

The `<spinner>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *decimalplaces* = `CDATA`

  Specifies the level of precision of *value*. For example, setting *decimalplaces* to 2 allows *value* to be specified with 2 digits to the left of the decimal point.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *editable* = `true` | `false`

  Default is `false`. If `true`, the number displayed in the control can be typed in directly. If `false`, the number can only be changed using the up and down arrows.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *increment* = *CDATA*

  Specifies the value by which the control can be increased and decreased.

- *maximum* = *CDATA*

  Specifies the lowest number in the control's range.

- *minimum* = *CDATA*

  Specifies the highest number in the control's range.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal resizing.

- *statustext* = *CDATA*

Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *value* = *CDATA*

  Specifies the default value of the control.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<tabbox> Element

The `<tabbox>` element creates a tabbed box control that has multiple boxes (container) or pages. The element can have the following child elements:

`<script>`, `<tabpanel>`

The `<tabbox>` element has the following attributes:

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal and vertical resizing.

- *selection* = *CDATA*

  Specifies the currently displayed page by ID.

- *width* = *CDATA*

Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<tablecontrol> Element

The `tablecontrol` element creates a multi-column list with column headers. The element can have the following child elements:

`<header>`, `<row>`, `<script>`

The `tablecontrol` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *columns* = *CDATA*

  Specifies the number of columns.

- *contextmenu* = *CDATA*

  The id or name of a shortcut (context) menu for this control. The value can be specified in an *id* attribute of a `<popupmenu>` element, or it can be the name of a shortcut menu loaded from a menu configuration file.

  If this attribute has no value, or if the value is not the id of a menu defined in the XUI dialog box file or that of a name loaded from a menu configuration file, no menu is displayed.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *gridlines* = `true` | `false`

  Default is `false`. If `true`, the table will have visible grid lines. If `false`, no gird lines will be displayed.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *nosortheader* = `true` | `false`

  Default is `false`. If `true`, clicking on the column heading will not resort the column contents.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal and vertical resizing.

- *showimages* = `true` | `false`

  Default is `false`. If `true`, images are displayed to the left of each `row` with an image defined.

- *sortedcolumn* = *CDATA*

  Number specifying the column to use when sorting rows. Setting *sortedcolumn* to `0` sorts on the first column. Setting *sortedcolumn* to `1` sorts on

the second column, and so on. Setting *sortedcolumn* to `-1` disables column sorting.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *type* = `single` | `multiple` | `cell`

  Default is `single`. If `single`, only one row can be selected in the table at one time. If `multiple`, more than one row can be selected in the table. If `cell`, clicking in the table selects a single cell and sets the `cell` element's *selected* attribute to `true`.

- *value* = *CDATA*

  Specifies the text in the first column of the selected row.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<tabpanel> Element

The `<tabpanel>` element creates a tabbed panel control that is a box (or page) within a tabbed box. The element can have the following child elements:

`<box>`, `<button>`, `<checkbox>`, `<colordropdown>`, `<combobox>`, `<description>`, `<grid>`, `<groupbox>`, `<label>`, `<listbox>`, `<morph>`, `<picturebox>`, `<radio>`, `<radiogroup>`, `<script>`, `<separator>`, `<slider>`, `<spacer>`, `<spinner>`, `<tabbox>`, `<tablecontrol>`, `<textbox>`, `<treecontrol>`, `<unitdimensionbox>`

The `<tabpanel>` element has the following attributes:

- *align* = `start` | `center` | `end`

  Default is `start`. Specifies how space not used by the children is laid out. `start` places space below children. `center` evenly distributes space above and below children. `end` places space above children.

- *childrensize* = `vary` | `equalwidth` | `equalheight` | `equal`

  Specifies whether the children of this control will have the same height and width as each other. Default is `vary`. If `vary`, the height and width of the

children will not be restricted. If `equalwidth`, the width of the children will be the same as that of the widest child. If `equalheight`, the height of the children will be the same as that of the tallest child. If `equal`, the width of the children will be the same as that of the widest child and the height of the children will be the same as that of the tallest child..

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *id* = *ID*

  Identifies the control.

- *label* = *CDATA*

  Specifies the text to display next to the control.

- *orient* = `vertical` | `horizontal`

  Default is `horizontal`. Specifies the layout for the container children.

- *pack* = `start` | `center` | `end` | `spread` | `stretch`

  Default is `spread`. Specifies how space not used by the children is laid out. `start` places space after children. `center` evenly distributes space before and after children. `end` places space before children. `spread` evenly distributes space before, between, and after children. `stretch` evenly distributes space between children, with no space before or after children.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal and vertical resizing.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

<textbox> Element

The `<textbox>` element creates text box and multi-line text box controls. The element can have the following child elements:

`<script>`, `<value>`

The `<textbox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`,

`gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.

- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.

- *id* = *ID*

  Identifies the control.

- *maxlength* = *CDATA*

  Specifies the maximum amount of characters within the edit field.

- *multiline* = `true` | `false`

Default is `false`. If `true`, the edit field may wrap.

- *readonly* = `true` | `false`

  Default is `false`. If `true`, the current value of the item cannot be modified.

- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction.

  If `natural`, the control is horizontally resizeable. If *multiline* = `true`, the control is also vertically resizeable.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *type* = `normal` | `password`

  Default is `normal`. If `normal`, the content of the `value` child element is displayed as entered. If `password`, the content of the `value` child element is displayed as asterisks.

- *wantreturn* = `true` | `false`

  (Multi-line text boxes only.) Default is `true`. If `true`, a carriage return is inserted when the user presses the **Enter** key while entering text. If `false`, pressing the **Enter** key has the same effect as clicking on the dialog box's default control. This value is ignored when set for single-line text boxes.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<toolbar> Element

The `<toolbar>` element is the parent element for creating a collection of tool bar buttons. The element can have the following child elements:

`<button>`, `<checkbox>`, `<colordropdown>`, `<combobox>`, `<listdropdown>`, `<script>`, `<separator>`, `<textbox>`

The `<toolbar>` element has the following attributes:

- *dock* = `bottom` | `left` | `none` | `right` | `top`

  Default is `top`. Specifies the edge used to dock the tool bar.
- *enabledocking* = `any` | `bottom` | `bottomleft` | `bottomleftright` | `bottomright` | `left` | `leftright` | `right` | `top` | `topbottom` | `topbottomleft` | `topbottomright` | `topleft` | `topleftright` | `topright`

  Default is `any`. Specifies which edges allow docking.
- *id* = *ID*

  Identifies the control.
- *name* = *CDATA*

  Text used to display in the **View** ▸ **Toolbars** menu list. For example, **Edit** or **Table**. *name* can also be referenced by scripts to access the `toolbar` object.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.
- *x* = *CDATA*

  A number specifying the horizontal screen coordinate to position the upper left corner of the dialog box.
- *y* = *CDATA*

  A number specifying the vertical screen coordinate to position the upper left corner of the dialog box.

<toolbargroup> Element

The `<toolbargroup>` element creates a container for multiple tool bars. The element can have the following child element:

`<toolbar>`

The `<toolbargroup>` element has the following attribute:

- *id* = *ID*

  Identifies the control.

<treecontrol> Element

The `<treecontrol>` element creates an outline list of items (defined by `<treenode>` elements). The element can have the following child elements:

`<script>`, `<treenode>`

The `<treecontrol>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *branchimage* = *IDREF*

  Specifies by reference a graphical image to be displayed for a branch in the tree. The value of *branchimage* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *contextmenu* = *CDATA*

  The id or name of a shortcut (context) menu for this control. The value can be specified in an *id* attribute of a `<popupmenu>` element, or it can be the name of a shortcut menu loaded from a menu configuration file.

  If this attribute has no value, or if the value is not the id of a menu defined in the XUI dialog box file or that of a name loaded from a menu configuration file, no menu is displayed.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *extraimage* = *IDREF*

  Specifies by reference a second graphical image to be displayed for a branch or leaf in the tree. The value of *extraimage* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.

- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.

- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.
- *haslines* = `true` | `false`

  Default is `true`. If `true`, dotted lines will be displayed among tree nodes to indicate the hierarchy of tree nodes.
- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *leafimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for a leaf in the tree. The value of *leafimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the default MFC image is displayed.
- *openbranchimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for an open branch in the tree. The value of *openbranchimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the default MFC image is displayed.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal and vertical resizing.
- *selectedimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for a selected item in the tree. The value of *selectedimage* matches the value of the `image`

element *id* attribute specifying the desired graphic file. If not specified, the default MFC image is displayed.

- *seltype* = `single` | `multiple`

  Default is `single`. If `single`, only one only one tree node can be selected at a time. If `multiple`, more than one tree node can be selected at one time.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *type* = `normal` | `checkboxes` | `radiobuttons`

  Default is `normal`. If `checkboxes`, a check box will be displayed with each node. If `radiobuttons`, a radio button control will be displayed with each node. If `normal`, only the node will be displayed.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<treenode> Element

The `<treecontrol>` element creates a node (branch) in a `<treecontrol>` list. The element can have the following child elements:

`<script>`, `<treenode>`

The `<treecontrol>` element has the following attributes:

- *appdata* = *CDATA*

  Specifies a value for later reference.

- *branchimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for a branch in the tree. The value of *branchimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the image specified by `treecontrol` is displayed.

- *checkstate* = `checked` | `unchecked` | `indeterminate`

  Default is `unchecked`. If `checked`, the item is selected. If `unchecked`, the item is not selected. If `indeterminate`, the item displays no state.

- *checkstyle* = `none` | `checkbox` | `radiobutton`

Default is `none`. If `checkbox`, the tree node has a check box. If `radiobutton`, the tree node has a radio button. If `none`, the tree node has no check box or radio button.

- *expanded* = `true` | `false`

  Default is `false`. If `true`, the node is expanded to display the node's children.

- *extraimage* = *IDREF*

  Specifies by reference a second graphical image to be displayed for a branch or leaf in the tree. The value of *extraimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the image specified by `treecontrol` is displayed.

- *id* = *ID*

  Identifies the control.

- *label* = *CDATA*

  Specifies the text to display within the control.

- *leafimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for a leaf in the tree. The value of *leafimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the image specified by `treecontrol` is displayed.

- *openbranchimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for an open branch in the tree. The value of *openbranchimage* matches the value of the `image` element *id* attribute specifying the desired graphic file.

- *selected* = `true` | `false`

  Default is `false`. If `true`, *selected* is the currently selected node. Only one node can be selected at any time.

- *selectedimage* = *IDREF*

  Specifies by reference the graphical image to be displayed for a selected item in the tree. The value of *selectedimage* matches the value of the `image` element *id* attribute specifying the desired graphic file. If not specified, the image specified by `treecontrol` is displayed.

- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

<unit> Element

The `<unit>` element specifies a valid unit of measure for the parent `<unitdimensionbox>`. The element can have no child elements.

The `<unit>` element has the following attributes:

- *id* = *ID*

  Identifies the control.

- *label* = *CDATA*

  String specifying the unit of measure. For example, `pt`, `in`, `cm`, `lb`, `kg`, and so on.

<unitdimensionbox> Element

The `<unitdimensionbox>` element creates a control used for specifying a value by entering the value in an edit field or by changing the current value using up and down arrows. The units of measure are displayed in the edit field with the value. The element can have the following child elements:

`<script>`, `<unit>`

The `<unitdimensionbox>` element has the following attributes:

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's background. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *decimalplaces* = `CDATA`

  Specifies the level of precision of *value*. For example, setting *decimalplaces* to 2 allows *value* to be specified with 2 digits to the left of the decimal point.

- *disabled* = `true` | `false`

  Default is `false`. If `true`, the control ignores all user interface events directed toward the control.

- *family* = *CDATA*

  Specifies the font family.

- *fontposture* = `italic` | `upright`

  Default is `upright`. Specifies the control's font posture.

- *fontsize* = *CDATA*

  Specifies the control's font size in points.

- *fontstyle* = `monsanserif` | `monoserif` | `sanserif` | `serif`

  Default is `sanserif`. Specifies the control's font style.
- *fontweight* = `bold` | `medium`

  Default is `medium`. Specifies the control's font weight.
- *foregroundcolor* = *CDATA*

  Specifies the color to use in drawing the control's foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.
- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *helpid* = *CDATA*

  Number specifying the help topic to display for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *increment* = *CDATA*

  Specifies the value by which the control can be increased and decreased.
- *maximum* = *CDATA*

  Specifies the lowest number in the control's range.
- *minimum* = *CDATA*

  Specifies the highest number in the control's range.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control allows horizontal resizing.
- *statustext* = *CDATA*

  Specifies the text to display in status windows.
- *tiptext* = *CDATA*

  Specifies the text to display as context-sensitive help for this control.

- *unit* = *CDATA*

  Specifies the default unit of measure of the control. If not provided, `pt` is used.
- *value* = *CDATA*

  Specifies the current value of the control.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<value> Element

The `<value>` element is a child element containing a value. The element can have no child elements.

The `<value>` element has the following attribute:

- *id* = *ID*

  Identifies the control.

<webview2> Element

The `<webview2>` element is a container for an WebView2control.

The element can have the following child elements:

`<script>`

The `<webview2>` element has the following attributes:

- *height* = *CDATA*

  Number specifying the height in pixels to allocate for this control.
- *hidden* = `true` | `false`

  Default is `false`. If `true`, the space for the control is allocated, but the control itself is not displayed.
- *id* = *ID*

  Identifies the control.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows

resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.

- *statustext* = *CDATA*

  Specifies the text to display in status windows.

- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.

- *withdraw* = `true` | `false`

  Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

<window> Element

The `<window>` element is the top level element of a XUI dynamic dialog box. It represents the window frame.

The element can have the following child elements:

`<activex>`, `<box>`, `<button>`, `<checkbox>`, `<colordropdown>`, `<combobox>`, `<description>`, `<grid>`, `<groupbox>`, `<imagegroup>`, `<label>`, `<listbox>`, `<menubar>`, `<morph>`, `<picturebox>`, `<radio>`, `<radiogroup>`, `<script>`, `<separator>`, `<slider>`, `<spacer>`, `<spinner>`, `<tabbox>`, `<tablecontrol>`, `<textbox>`, `<toolbargroup>`, `<treecontrol>`, `<unitdimensionbox>`

The `<window>` element has the following attributes:

- *align* = `start` | `center` | `end`

  Default is `start`. Specifies how space not used by the children is laid out. `start` places space below children. `center` evenly distributes space above and below children. `end` places space above children.

- *backgroundcolor* = *CDATA*

  Specifies the color to use in drawing the window background. Values can be the standard HTML named colors, the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`, or `transparent`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

  A value of `transparent` specifies that embedded dialog boxes are to have no background color.

- *childrensize* = `vary` | `equalwidth` | `equalheight` | `equal`

Specifies whether the children of this control will have the same height and width as each other. Default is `vary`. If `vary`, the height and width of the children will not be restricted. If `equalwidth`, the width of the children will be the same as that of the widest child. If `equalheight`, the height of the children will be the same as that of the tallest child. If `equal`, the width of the children will be the same as that of the widest child and the height of the children will be the same as that of the tallest child..

- *dock* = `none` | `bottom` | `left` | `right` | `top`

  Specifies the docking state of the dockable dialog box. The default value of *dock* is `none`. The value of *dock* must be one of the locations specified by *enabledocking*. Otherwise, *dock* will be ignored.

  The *modal* attribute in the `<window>` element takes precedence over the *enabledocking* attribute. If *modal* has a value of `true`, the XUI file will be displayed as a standard modal dialog box regardless of the value of *enabledocking*.

- *enabledocking* = `none` | `any` | `bottom` | `left` | `right` | `top` | `topbottom` | `topleft` | `topright` | `bottomleft` | `bottomright` | `leftright` | `topbottomleft` | `topbottomright` | `topleftright` | `bottomleftright`

  Specifies the edges of the edit window the dialog box can dock to. The default value of *enabledocking* is `none`. If *enabledocking* is `none`, the XUI file will be displayed as a non-dockable (standard) dialog box.

- *focus = IDREF*

  Specifies by ID the element currently with focus.

- *foregroundcolor = CDATA*

  Specifies the color to use in drawing the window foreground. Values can be the standard HTML named colors plus the Arbortext colors `gray1`, `gray2`, `gray3`, `gray4`, `gray5`. Colors can also be a string of a `#` followed by three two-digit hexadecimal numbers specifying the red, green, and blue (RGB) intensity values defining a color.

- *height = CDATA*

  Number specifying the height in pixels to allocate for this control.

- *helpfile = CDATA*

  Specifies the path and file name of the Microsoft HTML Help file containing help for this dialog box.

- *id = ID*

  Identifies the control.

- *margin* = *CDATA*

  Default is `12`. Specifies the margin around the content of the dialog box in pixels.

  For example, if the value of *margin* is `6`, each of the left, right, top, and bottom margins of the dialog box will be 6 pixels. If the value is `0`, the dialog box will have no margins. If *margin* is not specified each margin will be 12 pixels.
- *modal* = `true` | `false`

  Default is `false`. If `true`, the dialog box must be acknowledged before other windows can be accessed.
- *orient* = `vertical` | `horizontal`

  Default is `horizontal`. Specifies the layout for the container children.
- *pack* = `start` | `center` | `end` | `spread` | `stretch`

  Default is `spread`. Specifies how space not used by the children is laid out. `start` places space after children. `center` evenly distributes space before and after children. `end` places space before children. `spread` evenly distributes space before, between, and after children. `stretch` evenly distributes space between children, with no space before or after children.
- *resize* = `none` | `both` | `height` | `width` | `natural`

  Default is `natural`. If `width`, the control allows horizontal resizing only. If `height`, the control allows vertical resizing only. If `both`, the control allows resizing in both directions. If `none`, the control doesn't resize in either direction. If `natural`, the control resizes if necessary depending on size changes in its children.
- *status* = *IDREF*

  Specifies the id of the control displaying the *statustext* value of the control with focus.
- *title* = *CDATA*

  The text displayed in the frame title bar.
- *width* = *CDATA*

  Number specifying the width in pixels to allocate for this control.
- *windowname* = *NMTOKEN*

  Text to identify the window.
- *withdraw* = `true` | `false`

Default is `false`. If `true`, the control (and any children) is ignored and not displayed.

- *xmlns:ev = CDATA*

  Specifies the XML events namespace URL.

7

Working with ActiveX Controls

Overview

Microsoft ActiveX controls are Windows-only objects that combine features of COM servers, COM Automation, COM Event interfaces, and ActiveScript hosting. ActiveX controls are self-contained applets or controls which expose their functionality to third-party programmers through COM. Arbortext Editor lets you configure, launch, and use these COM controls to extend Arbortext Editor with tools from a variety of software vendors or through using those controls you create yourself. You can also run Arbortext Editor itself in an ActiveX control.

Using ActiveX controls provide the following benefits:

- Less programming — When using existing controls, developers need only be concerned with writing scripts that handle the communication of data between their XML and the controls. They do not need to design the user interface, develop the user interface, or design the API or properties that hold the data entered by the user.
- Enhanced end-user productivity — Create specialized user interfaces for particular types of documents.
- Simplified complex tagging operations — Create boilerplate sections of documents, set sections to be read-only, create tagging with desired attributes or elements completed, or otherwise control the end-users' interaction with documents.

For details on creating and using ActiveX controls in general, refer to the ActiveX Controls section of Microsoft's MSDN Library. (http://msdn.microsoft.com/library/) and numerous other locations on the World Wide Web. The remainder of this chapter covers configuring and launching ActiveX controls with Arbortext Editor and running Arbortext Editor in an ActiveX control.

Arbortext Editor and ActiveX Controls

ActiveX controls can be implemented with Arbortext Editor in two ways:

- In a separate XUI file specifying the operation of the control. Refer to the description of the activex element in the *Working with XUI (XML-based User Interface) dialog boxes* chapter of the *Customizer's Guide*.
- Defined and called directly from the `.dcf` file. The control is still launched in a generated XUI dialog box. Refer to Executing ActiveX Controls Using the .dcf File to Bind to an Element Directly on page 218.

Keep the following items in mind when working with ActiveX controls and Arbortext Editor.

- Using ActiveX controls with Arbortext Editor is supported on Windows platforms. Refer to the *Installation Guide for Arbortext Editor, Arbortext*

Styler, and Arbortext Architect for a detailed list of requirements for working with ActiveX controls.

- Rhino JavaScript does not support ActiveX controls in Arbortext Editor.
- ActiveX controls are third-party tools. Many free controls are available to Windows users, as are many commercially licensed controls. Licensing and obtaining proper documentation for these third-party products is your responsibility. No claims are made as to the reliability or fitness of a third-party control or its compatibility with Arbortext Editor.

Running Scripts

When working with ActiveX controls, keep the following items in mind:

- To call an ActiveScript from another ActiveScript, use the `Application.createScriptContext` and `ScriptContext.loadScriptFile` methods. Do not use the ACL `source` command from within a JScript or VBScript file as in:
  ```
  Acl.execute("source test.js");
  ```
  Calling the `source` command from an ActiveScript will report an error message in the parent script if the child script contains errors. If both scripts are error free, the nesting will silently fail.
- Any session-level scripts placed in *`Arbortext-path`*`\custom\init` or any directory defined by the *APTCUSTOM* environment variable will be executed automatically when Arbortext Editor starts.
- Any document type level scripts (such as *`Arbortext-path`*`\custom\doctypes\axdocbook\axdocbook.js`) will be run automatically the first time a document of the named document type is opened or created. The *`doctype`*`.*` file is not run on subsequent openings of files of the same document type.
- Any document-level scripts (such as *`Arbortext-path`*`\custom\doctypes\axdocbook\instance.js`) will be run automatically each time a document of the named document type is opened or created.
- Any instance-specific scripts (such as *`filename`*`.vbs` where *filename* is the same root name as that of the document) will be run each time a specific document of that name is opened or created.

Related AOM Interfaces

Working with ActiveX controls makes use of the following AOM interfaces. (Refer to the *Programmer's Reference* for details on all AOM interfaces.)

Interface	Description
Application	The `createScriptContext` method creates an object that can be used to load, compile, and execute scripts using the Microsoft Windows Script engine. The `getScriptContext` method returns a pointer to a **ScriptContext** object for a running script.
ScriptContext	Provides the methods necessary to load and run scripts using the Microsoft Windows Scripting engine.

Executing ActiveX Controls Using XUI

You can implement ActiveX controls using the XUI `<activex>` control. (You can also implement ActiveX controls directly from the `.dcf` file. That implementation is detailed in Executing ActiveX Controls Using the .dcf File to Bind to an Element Directly on page 218.) The `<activex>` element is fully defined in the Working with XUI (XML-based User Interface) dialog boxes chapter of the *Customizer's Guide*, and can be described using XUI as in the following example:

```
<activex progid = "MSCAL.Calendar" id = "date">
</activex>
```

The *progid* attribute specifies which ActiveX control is to be added to the dialog box. It can either be a Program ID for the control, as shown in the previous example, or the value of the control's CLSID property as a string surrounded by curly brackets as in the following example:

```
<activex progid="{2398E32F-5C6E-11D1-8C65-0060081841DE}" id="TextToSpeach">
</activex>
```

The *id* parameter identifies the control.

Using the `<param>` child tag of the `<activex>` element allows you to provide parameters name-value pairs for the ActiveX control. For example:

```
<activex progid="MSCAL.Calendar" id="date">
 <param name="ShowDateSelectors" value="false"/>
</activex>
```

By setting the parameter `ShowDateSelectors` to `false`, the date selector dropdowns (one for the month and another for the year) are removed from the Calendar ActiveX control.

An ActiveX control can be launched embedded in the document displayed in Arbortext Editor, as a standalone control, or as a control within a XUI dialog box. When launched as a standalone control, a XUI dialog box is automatically generated to contain the control.

Example: Embedded Calendar Control

This example uses XUI technology to embed the Microsoft Calendar control in a document based on the Arbortext XML DocBook (axdocbook) DTD and displayed in Arbortext Editor. When the user selects a date, the date is added as the content of the document's `<date>` element. Using the Calendar control for entering the date eliminates the possibility of the user mis-typing the date. The Calendar control also provides a well-known user interface for working with dates.

Implementation

Use the following steps to implement this example:

1. Specify the control to call and its layout with the following markup saved as `date.xml`:

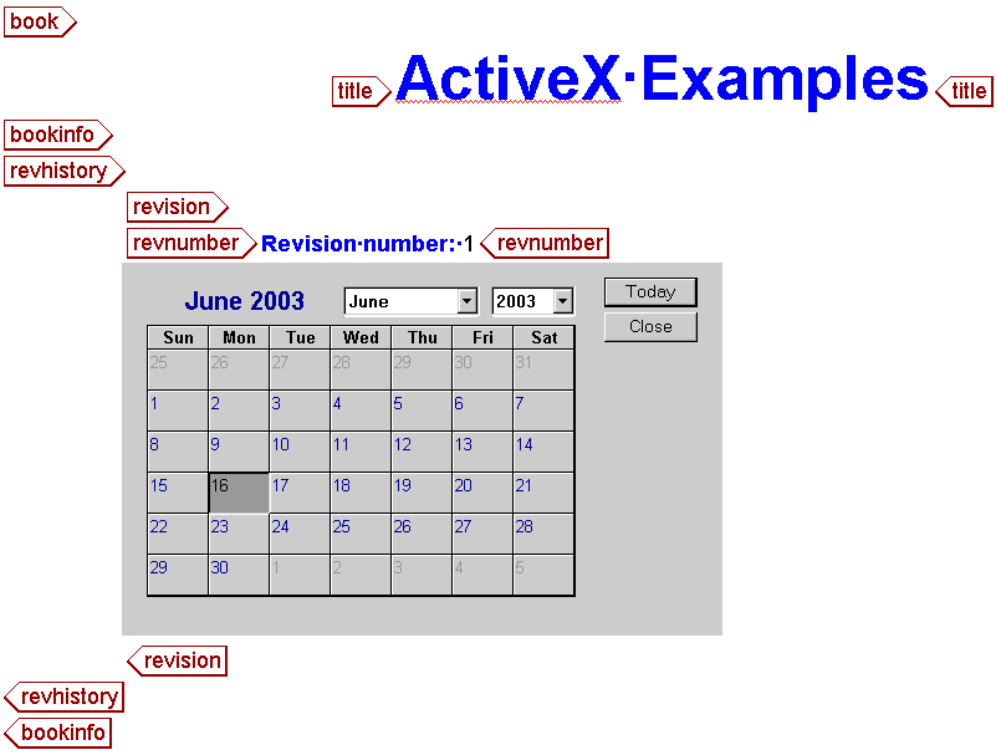
```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.0//EN"
 "xui.dtd">
<window width="400" height="300" orient="horizontal"
 focus="todayButton" modal="false" title="Test ActiveX"
 xmlns:ev="http://www.w3c.org/2001/xml-events">
<activex id="date" progid="MSCAL.Calendar">
 <script type="application/x-jscript" ev:event="Value_OnChange"
 ev:target="date">
 var node = Application.activeDocument.getElementById("date");
 if (node.hasChildNodes()) {
 node.firstChild.data = date.Value;
}
 else {
 var text = node.ownerDocument.createTextNode(date.Value);
 node.appendChild(text);
}
 </script>
</activex>
<spacer resize="none" width="4"/>
<box resize="none" orient="vertical" pack="end">
<button id="todayButton" label="Today">
 <script type="application/x-jscript" ev:event="domactivate">
 date.Today();
 </script>
</button>
<button label="Close" type="cancel">
```

```
 <script type="application/x-jscript" ev:event="domactivate">
 var dialog = Application.event.view.window;
 dialog.close();
 </script>
</button>
</box>
<script type="application/x-jscript" ev:event="windowload">
date.Today();
</script>
</window>
```

2. Configure the `.dcf` file to assign the file `date.xml` to the `<date>` element:

```
<Specials>
<XuiControl element="date" xuiFileName="date.xml"/>
```

3. Open a document that uses the DTD for which you've configured the `.dcf` file and insert a `<date>` element. For example, using the default axdocbook stylesheet, the following markup renders in Arbortext Editor as.



Calendar control embedded in a document

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE book PUBLIC "-//Arbortext//DTD DocBook XML V4.0//EN"
 "axdocbook.dtd">
<?Pub Inc?>
<book>
```

```
<title>ActiveX Examples</title>
<bookinfo>
<revhistory>
<revision>
<revnumber>1</revnumber>
<date id="date"></date>
</revision>
</revhistory>
</bookinfo>
...
</book>
```

Example: Previewing Word and Excel Documents

This example demonstrates how ActiveX dialog boxes can be implemented at a script level. In this scenario, users need to be able to preview Word and Excel documents from within Arbortext Editor. Because this type of functionality is not document type specific, the file could be activated from a custom Arbortext Editor menu item.

The following file supports this example and is available at the following location: *Arbortext-path*\samples\activex\commdlg.xml

This example launches the Microsoft Common Dialog control, an ActiveX version of the standard **File Open** dialog box. The script launches Microsoft Excel or Word depending on the user's selection on the **File Open** dialog box. The only ActiveX piece of the implementation is the **File Open** dialog box. Launching of Word and Excel uses standard ACL COM functions.

```
<!--ArborText, Inc., 1988-2003, v.4002-->
<!DOCTYPE window PUBLIC "-//Arbortext//DTD XUI XML 1.0//EN"
 "xui.dtd">
<window orient="horizontal" focus="opendlg" modal="false"
xmlns:ev="http://www.w3c.org/2001/xml-events">
<activex id="opendlg" progid="MSComDlg.CommonDialog"></activex>
<script type="application/x-vbscript"
 ev:event="windowload"
 ev:propagate="continue">
' Set flags
OpenDlg.Flags = cdlOFNHideReadOnly
' Set filters
OpenDlg.Filter = "Excel Files(*.xls)|*.xls|" & _
"Word Documents (*.doc)|*.doc|CSV Files(*.csv)|*.csv|Text Files" & _
"(*.txt)|*.txt|All Files (*.*)|*.*|"
' Specify default filter
OpenDlg.FilterIndex = 1
' Display the Open dialog box
opendlg.ShowOpen
index = OpenDlg.FilterIndex
filename = OpenDlg.filename
Application.ActiveWindow.Close()
```

```
if(filename <> "") then
 ' Let's launch Excel or Word as COM servers
 ' (Not to be confused with the ActiveX control from which
 ' we selected a file name.)
 if(index = 1 or index = 3) then ' if .xls or .csv
 ' Open file in Excel
 Dim hExcel 'handle to the Excel app
 Dim hWorkbooks 'handle to Excel Workbooks object
 set hExcel = CreateObject("Excel.Application")
 hExcel.Visible = -1
 set hWorkbooks = hExcel.Workbooks
 hWorkbooks.Open(filename)
 else ' for all other file types....
 'Open file in MS Word
 Dim hWord 'handle to the Word app
 Dim hWordDocs
 set hWord = CreateObject("Word.Application")
 hWord.Visible = -1
 set hWordDocs = hWord.Documents
 hWordDocs.Open(filename)
 end if
end if
</script>
</window>
```

Executing ActiveX Controls Using the .dcf File to Bind to an Element Directly

As an alternative to using an external XUI file to define the display of an ActiveX control, you can associate ActiveX controls with elements directly in the `.dcf` file, causing the controls to execute when triggered by specific DOM events. To use this alternate method to execute ActiveX controls from elements, you need to update your `.dcf` file to associate the element with the ActiveX control and related script. You also need to establish the proper element-to-control binding in the script.

Keep the following items in mind when using this method to associate ActiveX controls with elements:

- Arbortext Editor tracks the element-control relationships by element name. Therefore, a document can contain only one instance of a given element and its associated ActiveX control. For example, a document may contain numerous instances of an element that is associated with an ActiveX control, but only one ActiveX control can be displayed at any given moment.
- Each element must be associated with one script, creating a configuration trio of one element name to one script name to one control. A given control may be used multiple times, as long as each instance resides in its own

configuration trio. A given script file can be used multiple times, as long as the script name (as specified in the `ActiveXControl` element), is unique in each trio.

Configuring the .dcf File

To associate an element with an ActiveX control, edit the attributes of the `ActiveXControl` element in your custom `.dcf` file for the document type containing the element.

The `ActiveXControl` element has the following attributes:

Attribute	Description
`element`	(Required) Name of the XML or SGML element to be associated with the ActiveX control.
`controlName`	(Required) User-defined name for the control. Provides the link to event binding and script associations
`scriptFileName`	(Required) The system file name for the script file containing the required functions for handling the control. The current directory will be searched first for the script. The load path, containing the *`Arbortext-path`*`\custom\scripts` directory, will then be searched.
`scriptName`	(Optional) User-defined name for the script context. If not specified, a unique name for the script context is generated. By default, no two ActiveX controls created using the `.dcf` file will use the same script context.
`programId`	(Required) The name for the ActiveX control as registered in the Windows registry. It is the user-friendly form of the COM 128–bit Class ID (or GUID).
`scriptLanguage`	(Optional) Name by which users specify a scripting language which is not automatically recognized by file extension. VBScript and JScript are the only supported script languages.

Attribute	Description
`condition`	(Optional) An XPath expression. This attribute value allows the user to define more specific conditions for when an ActiveX control should be initiated for the given element. For example, to launch a particular ActiveX control based on a particular attribute value on a `graphic` element, that condition can be defined as an XPath expression. If the expression, in the context specified by the element instance, is `TRUE`, ActiveX processing will continue. If the expression is `FALSE`, the element will not cause any ActiveX behavior.
`inPlaceActivate`	(Optional) If `yes`, embeds the XUI dialog box containing the control in the document displayed in Arbortext Editor. If `no` (the default), displays the control in a standalone dialog box.
`eventName`	(Optional) The DOM event name that runs the ActiveX control. If unspecified, the default is `DOMActivate`.

As an example, a `.dcf` file's `ActiveXControl` element could have the following attributes:

```
<ActiveXControl
 element="date"
 controlName="cal"
 scriptFileName="date.js"
 scriptName="date"
 programId="MSCAL.Calendar"
 eventName="DOMActivate"
 inPlaceActivate="no">
</ActiveXControl>
```

This example specifies the following association:

- Run the control when a *DOMActivate* event happens on a `date` element.
- Create a new Microsoft ActiveX Calendar control with a `Program ID` of `MSCAL.Calendar`.

- Name the control "`cal`" in the script engines list of named variables. (This provides the name used for script event binding and the methods which write to and from the XML data, described below.)
- If the control attached to this DOM element is running, bring it to the desktop foreground. If the control attached to this DOM element has not been launched, then create and display it.
- Run the `Cal_OnInitialize(`*`activenode`*`)` function. This function name is a combination of the control name "`Cal`", an underscore character, and "`OnInitialize`". *activenode* is the element causing the ActiveX control to be loaded.
- Once the user selects a date and closes the control, the function `Cal_OnClose(`*`activenode`*`)` captures the current date from the Calendar and inserts the `<date>` element's PCDATA.

  If the window is closed because the element is deleted, the control window is closed and the `Cal_OnCancel(`*`activenode`*`)` function (if one exists) runs.

Refer to the examples at the end of this section for other sample `ActiveXControl` attribute settings.

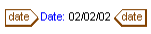
Establishing Element-to-Control Binding

Elements and ActiveX scripts are linked and bound to one another in the scripting environment with a naming convention that uses the control name specified in the `ActiveXControl` tag. Using the convention *controlname_function*, three script functions use this binding:

`controlname``_OnInitialize`(activenode)

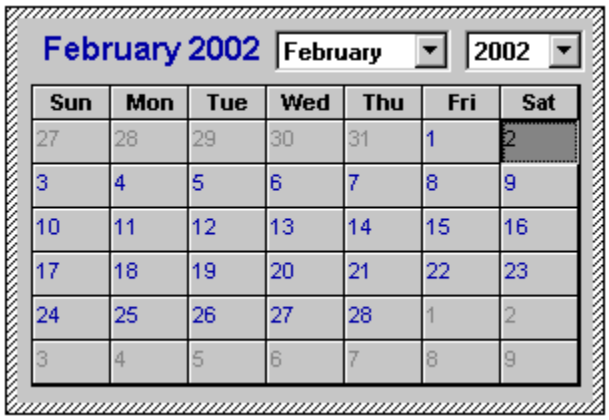
Performs all required control initialization, beyond that which might have been provided by any initialization file streams.

For example, the *`Cal`*`_OnInitialize(`*`activenode`*`)` function for a Calendar control might set the current date in the Calendar control according to any value in the current `<date>` element. *`Cal`*`_OnInitialize(`*`activenode`*`)` would retrieve the PCDATA:



The current date

and pass it to the *Calendar.Value* property to display as follows:



*controlname*_OnClose(activenode)

Modifies the XML data to reflect the final state of the ActiveX control. The function is not required, but is a logical way to synchronize the control data with the XML data. It is also a logical way to update the XML data in scripts that do not employ control event handlers.

When the user closes or cancels the dialog box, *controlname*_OnClose(*activenode*) is called if it exists in the active script. *controlname*_OnClose(*activenode*) is not called if the control is embedded in the document using inPlaceActivate.

*controlname*_OnCancel(activenode)

Performs any actions needed when Arbortext Editor closes a dialog box because its owner element was deleted. This function is optional, but can be useful to undo previous script actions in cases where XML data may have changed with control events. *controlname*_OnCancel(*activenode*) is not called if the control is embedded in the document using inPlaceActivate.

Example: Calendar Control

This example illustrates how to associate an ActiveX control with an element. Specifically, this example gives a means of entering dates into a document using the Microsoft Calendar Control.

The following files support this example and are available at the following locations:

- *Arbortext-path*\samples\activex\date.js

  A JScript sample file that associates the Microsoft Calendar Control with the <date> element for the Arbortext XML DocBook (axdocbook) DTD. The script handles all communication between the XML element and the control.

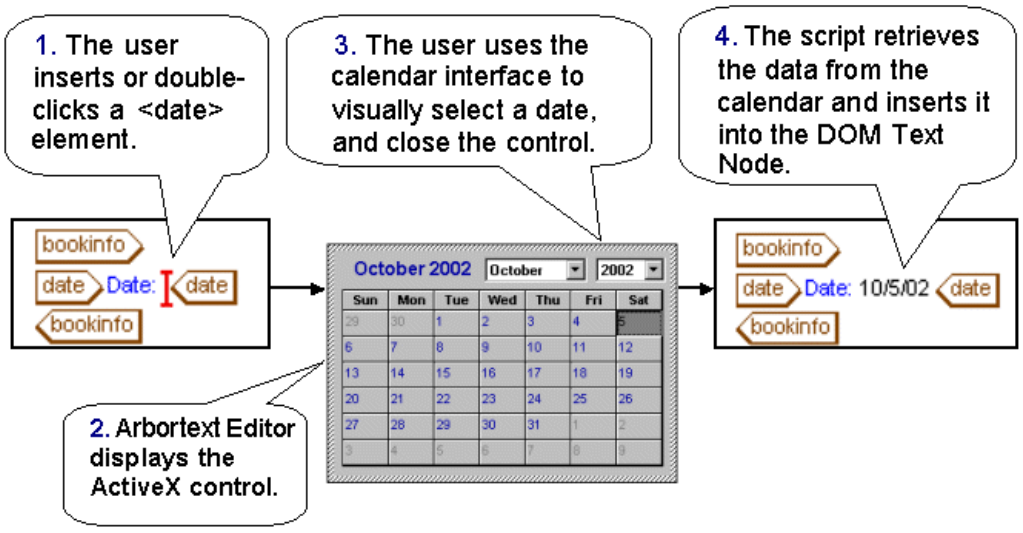
- *Arbortext-path*\samples\activex\date.vbs

  A VBScript version of date.js

Operation Overview

In this scenario, assume your users want to choose dates from the Calendar control interface instead of having to type the PCDATA into a `date` element.

The users interact with the Calendar control in the following manner:



First, the user inserts or double-clicks a `date` element. Second, Arbortext Editor displays the ActiveX control. Third, the user uses the calendar interface to visually select a date and close the control. Fourth, the script retrieves the data from the calendar and inserts it into the DOM Text Node.

Scripting Overview

From an implementation perspective, the following actions occur.

- When the user double-clicks the mouse or presses **shift-Enter** on a `<date>` element, a **DOMActivate** event is triggered.
- The script file `date.js` (which we named `date`) is executed.
- The Calendar control is displayed.
- The Calendar control is assigned the variable name *cal*. This lets the script have access to the control's methods and properties.
- The script method `cal_OnInitialize(ActiveNode)` is called.
- When the user closes the dialog box, the script method `cal_OnClose(ActiveNode)` is called.

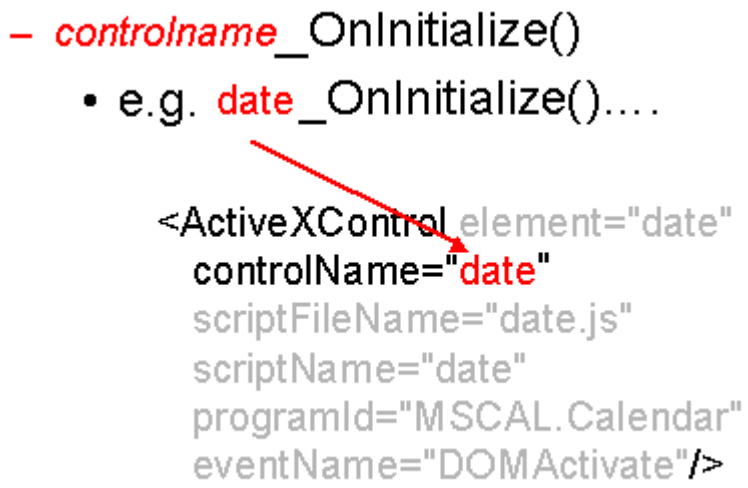
Implementation

Use the following steps to implement this example:

1. Configure the `.dcf` file by adding the following element:

```
<ActiveXControl element="date"
controlName="date"
scriptFileName="date.js"
scriptName="date"
programId="MSCAL.Calendar"
eventName="DOMActivate">
```

2. Create a control initialization function in your script. Match the function name prefix with the control name specified in the `ActiveXControl` element in the `.dcf` file as follows:



`ActiveXControl` element where the first part of the control name "`date_OnInitialize()`" parallels the `ActiveXControl` *controlName* attribute value of "`date`".

3. Populate the script initialization function as detailed in `date.js`:

```
function date_OnInitialize(ActiveNode)
{
 // Remember our active node for the embedded case
 anode = ActiveNode;
 // does this date element already have a child node?
 if(ActiveNode.HasChildNodes())
 {
 // yes, get the first child to see if it's our PCDATA
 var childNode = ActiveNode.FirstChild
 if(childNode.NodeType == NODE_TEXT)
 {
 // set the Calendar control to the value of the XML data
 date.Value = childNode.NodeValue
 date.Refresh()
}
}
```

```
}
```

4. Create and populate the script close function as detailed in `date.js`.

```
function date_OnClose(ActiveNode)
{
 // the active node should be an element,
 // but let's make sure...
 if(ActiveNode.NodeType == NODE_TEXT)
 {
 ActiveNode.replaceData(0, ActiveNode.length, date.Value);
}
 else if(ActiveNode.HasChildNodes())
 {
 // does this date element already have a child node?
 // and is it a text node?
 var childNode = ActiveNode.FirstChild
 if(childNode.NodeType == NODE_TEXT)
 {
 // yes, replace the text data with the control's data
 childNode.replaceData(0, childNode.length, date.Value);
}
}
 else
 {
 // no existing date PCDATA, create it
 insertDate(ActiveNode);
}
}
```

The close function calls the following `insertDate` function to insert the date selected in the control:

```
function insertDate(thisNode)
{
 // create a Range object
 var domRange = Application.ActiveDocument.createRange();
 // set that Range to the beginning of our <date> element
 domRange.setStart(thisNode,0);
 // select the entire <date> element
 domRange.selectNodeContents(thisNode);
 // collapse node to the end of the contents
 // i.e. the insertion point is just before the end tag </date>
 domRange.collapse(true);
 // create a new text node, and populate with the control data
 var textNode = Application.ActiveDocument.createTextNode(date.Value)
 // inserts the text.... <date>here</date>
 domRange.insertNode(textNode)
}
```

Running Arbortext Editor in an ActiveX Control

Besides running other ActiveX controls inside of Arbortext Editor, you can run Arbortext Editor itself in an ActiveX control. The Arbortext Editor ActiveX control is located at *Arbortext-path*\bin\EditorControl.dll. This control is registered as a COM server whenever the associated version of Arbortext Editor is registered.

Arbortext Editor can run as an embedded ActiveX control inside of another window or dialog box. In this case, the Arbortext Editor menus are made available through a new **Menu** toolbar button. Also, some menu choices and toolbar buttons are not available in an embedded frame.

Sample Arbortext Editor ActiveX implementations are in the *Arbortext-path*\samples\activex_editor directory. The XUI directory has a sample XUI dialog box containing an embedded Arbortext Editor ActiveX control. The CSharp directory contains a sample C# implementation. These examples demonstrate how to use the return value of the open method (which is an ACL window id) with the Acl.getWindow AOM method to obtain a Window AOM object. This technique provides the caller with great control over the window and the document living in it.

Characteristics of the ActiveX Arbortext Editor

Since the ActiveX version of Arbortext Editor is contained inside of some other window, it has somewhat different functionality than the regular Arbortext Editor window. The ActiveX Arbortext Editor has the following differences from the regular Arbortext Editor:

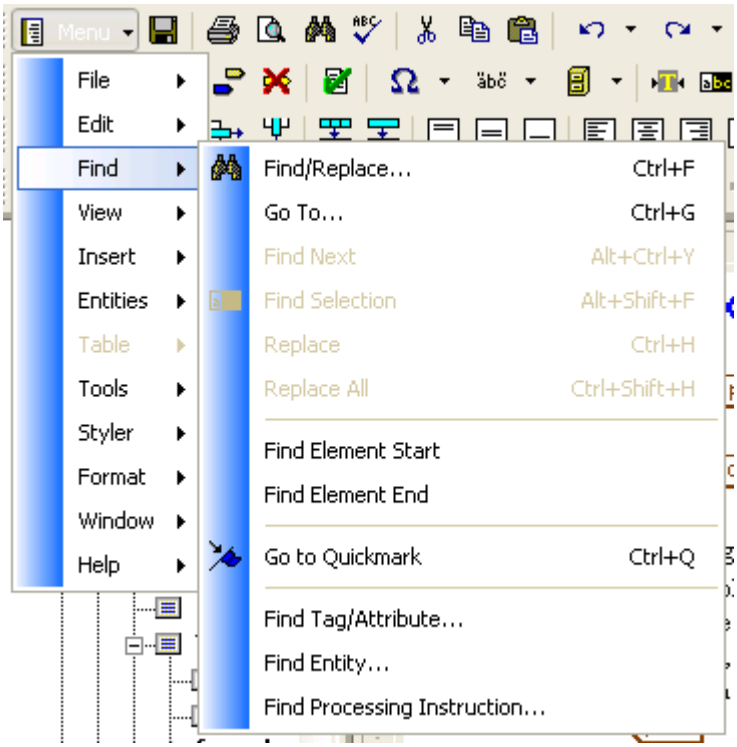
- No title bar — Only the window hosting the Arbortext Editor ActiveX control can have a title bar.
- Does not appear in the Microsoft Windows taskbar — Only the window hosting the Arbortext Editor ActiveX control can appear on the taskbar.
- Does not appear in the Microsoft Windows ALT+TAB window sequence — Only the window hosting the Arbortext Editor ActiveX control can appear in this sequence.
- Might not be resizable — The window hosting the Arbortext Editor ActiveX control determines whether the control can be resized.

  For example, if you embed the ActiveX control inside of a XUI dialog box, you can control whether the control is resized when the dialog box is resized.

The ActiveX Arbortext Editor has the following user interface differences from the regular window:

- The Arbortext Editor menus are available from a toolbar button.

The ActiveX Arbortext Editor provides the menus through a **Menu** toolbar button:



Menu toolbar button

In the ActiveX Arbortext Editor, the F10 key activates the **Menu** toolbar button and opens the menus.

- The following menu choices (and associated keyboard shortcuts) are not available:
  - **File ▸ New**
  - **File ▸ Open**
  - **File ▸ Close**
  - **File ▸ Save All**
  - **File ▸ Exit**
  - **File** menu list of recently opened documents
  - **Window ▸ New Window**
  - **Window ▸ Cascade**
  - **Window ▸ Default Window Configuration**
  - **Window** menu list of open Arbortext Editor windows

- All menu choices for editing FOSIs

- The following toolbar buttons are not available:
  - **New**
  - **Open**

The EditorControl.dll Control

The Arbortext Editor ActiveX control is contained in the `EditorControl.dll` file. The control has the following names when viewed with the Microsoft OLE/COM Object Viewer (`Oleview.exe`) application:

- `Arbortext.EditorControl` — The ProgID for the control
- `Arbortext Editor Control` — The name in the **Object Classes/All Objects** group
- `Arbortext Editor Control Type Library` — The name in the **Type Libraries** group
- `IArbortextEditorControl` — The name in the **Interfaces** group

Following is the Interface Definition Language (IDL) definition for the control's COM interface:

```
interface IArbortextEditorControl : IDispatch{
 [id(1), helpstring("Opens a new document after first closing the currently-displayed
 document (if any).")]
 HRESULT open(
 [in] BSTR documentPath,
 [in,defaultvalue(0)] LONG documentFlags,
 [in,defaultvalue(0)] LONG windowFlags,
 [in,defaultvalue("")] BSTR xuiPath,
 [out, retval] LONG* pWindowId
 );
 [id(2), helpstring("Displays an already-open document after first closing the
 currently-displayed document (if any).")]
 HRESULT show(
 [in] LONG documentId,
 [in,defaultvalue(0)] LONG windowFlags,
 [in,defaultvalue("")] BSTR xuiPath,
 [out, retval] LONG* pWindowId
 );
 [id(3), helpstring("Closes the currently-displayed document and, by default, prompts
 to save changes.")]
 HRESULT close([in,defaultvalue(0)] LONG flags);
};
```

HRESULT Return Values

Each method in the control's COM interface returns one of the following HRESULT values:

- S_OK — Operation successful
- 0x800704C7 — Operation canceled by user

  Some operations can display prompts that enable the user to cancel the operation. In this case, the control returns an HRESULT of 0x800704C7. Encoded in this result is a FACILITY code of FACILITY_WIN32 and a WIN32 error code ERROR_CANCELLED. This enables the calling code to determine the operation was canceled by the user. The associated message will be The operation was canceled.
- E_FAIL — Unexpected failure

  In this case, there will be an associated error message for the type of failure.

open Method

Opens a resource as a Arbortext Editor document and displays it in the ActiveX embedded Arbortext Editor window. This method first closes the current document (if any) and prompts the user to save it (if modified). Note that at the **Save** prompt, the user might **Cancel** the operation. In this case, the current document is left intact and this method returns an error.

If you do not want the user to see a **Save** prompt, then you can call the close method to close the current document explicitly before calling the open method. Note that you can suppress all informational windows during an open operation by specifying the 0x0020 value for the *documentFlags* parameter.

open(documentPath [, documentFlags [, windowFlags [, xuiPath]]])	
Parameters	String *documentPath* Represents the path to a document. This can be in any form that Arbortext Editor recognizes, such as with the edit ACL command or the Application.openDocument AOM method. For example: • c:\documents\guide.xml • http://server/documents/guide.xml • x-wc://file=84756484.xml int *documentFlags* [optional] Defaults to 0. A bitmask that specifies open options. Constructed by ORing the bits from the following enumeration:

	OPEN_RDONLY = 0x0001 Open for read only and do not lock the underlying file. If this is not set, the underlying file will be locked if possible and the document will be read-only if no lock was acquired. The “checked out” status of CMS Objects will not be affected. **OPEN_DOCRDWR = 0x0002** Open for writing and do not lock the underlying file. The document will be modifiable even though the underlying file is not locked. If the document was already open in memory, this will additionally attempt to lock the underlying file. The “checked out” status of CMS Objects will not be affected. **OPEN_NLOCK = 0x0004** Do not lock the underlying file. Overrides all other flags which might acquire a file lock. The resulting document will not be modifiable unless `OPEN_DOCRDWR` is also given. The “checked out” status of CMS Objects will not be affected. **OPEN_CC = 0x0008** Perform a completeness check when reading the SGML file. This option is ignored for XML documents. **OPEN_NOCC = 0x0010** Suppress the completeness check when reading the SGML file. This option is ignored for XML documents. `OPEN_NOCC` is the default option for SGML documents saved by Arbortext Editor. **OPEN_NOMSGS = 0x0020** Do not display any parser error messages in a message window. Instead, suppress all warnings and errors. **OPEN_XML = 0x0100** Open the document as an XML document even if it does not start with the XML version processing instruction. If not specified, the document is loaded as

	an SGML document unless the document starts with the XML version header. **OPEN_NOSTYLE = 0x0200** Open the document without loading a style sheet. **OPEN_NODTPROMPT = 0x0400** Do not prompt the user if the document type associated with the document instance does not exist or is not compiled. Instead, fail the operation. **OPEN_RECTABLES = 0x4000** Cause the table editor to recognize tables immediately after opening the document. By default, table objects are not created until the document is displayed in a window. **OPEN_EDITINIT = 0x8000** Process initialization files immediately after opening the document. This includes sourcing the associated document type instance files (`instance.acl`, `instance.js`, and `instance.vbs`) and the document command files (docname `.acl`, docname`.js`, and docname `.vbs`). By default, these files are not processed until the document is displayed in a window. **OPEN_NEW_DOC = 0x10000** Treat the document as if it were created using the `New` dialog box. In this case, the path name is set to null and the document name is of the form `DocumentN`. **OPEN_FREEFORM = 0x80000** Open the document in free form mode, ignoring the document type specified in the file. `int` *windowFlags* [optional] Defaults to 0. A bitmask that specifies window options. Constructed by ORing the bits from the following enumeration: • `0x00001` - Supply vertical scrollbar (pane). • `0x00002` - Supply menu bar. If this is set then the **Menu** toolbar button will be shown. If this is not set, the **Menu** toolbar button is not shown and there will be no way to bring up menus.

	The menu bar is part of the Edit toolbar (Toolbar 1) — 0x00020. Toolbar 1 must be supplied to enable the display of the menu bar. • `0x00004` - Supply command subwindow. • `0x00008` -Supply message footer subwindow. • `0x00020` -Supply the Edit toolbar (that is, Toolbar 1) (pane). • `0x00080` - Supply horizontal scrollbar (pane). • `0x00100` - Do `edit` command intializations, include reading the document type instance command files (`instance.acl` and `instance.js`) and document command files (docname`.acl` and docname `.js`) if they exist, and calling the ACL `editfilehook` when a document is attached to the window. This bit applies only to edit class windows (pane). • `0x01000` - Supply a table column width ruler (pane). • `0x02000` - Supply a table row height ruler (pane). • `0x04000` - Supply the Markup toolbar (that is, Toolbar 2). • `0x08000` - Supply the Table toolbar (that is, Toolbar 3).

	• `0x10000` - Supply the Application toolbar (that is, Toolbar 4). If a menu bar is requested, it must be initialized using the **menu_load** or **menu_add** ACL commands before the window is first displayed. If a message footer is created, error messages and output from the **message** ACL command are displayed in the left part of the footer if the message is short enough (otherwise a popup dialog box is used). Any messages directed to the message footer are considered transient and are erased on the next key or button event received in the window. If this parameter is omitted or zero, it behaves as if the following value were given: `0x9F9FF`. This corresponds to the internal ACL constants (*h::winMaskMain* | *h::winMaskEditStyle*) which are used to create default "edit" windows. Over time, these constant might change, so this ensures the latest defaults are used. Regardless of the value given, the following bits will be forcibly set:`0x80850`. This ensures that some window properties are set that are absolutely required for the control to work properly. To cause the document to be shown with an absolute minimum of window artifacts (toolbars, scroll bars, status bar, etc..), any non-empty subset of `0x80850` can be used. For example: `0x00040`. `String` *xuiPath* [optional] Defaults to an empty string. An optional parameter used to supply an alternative XUI file to define the toolbars used by the edit window. If *xuiPath* is not supplied (or empty), then *`Arbortext-path`*`\lib\dialogs\editwindow.xml` is used.
Returns	The ACL ID of the possibly new window that was loaded as a result of this call. A value of `-1` can be returned if no *documentPath* parameter was given.

Throws	A COM error will be returned to the caller for the following cases: • User canceled at the save prompt or at some prompt while opening the requested document. In this case, returns an HRESULT value of `0x800704C7` which enables the caller to distinguish this case from other types of failures. This represents a FACILITY of `FACILITY_WIN32` and a WIN32 error code of `ERROR_CANCELLED`. • Failed to open the requested resource. In this case. the `HRESULT` will be `E_FAIL` The associated COM error message is dependent upon the actual failure opening the requested resource.

show Method

Opens an existing resource based on its ACL document ID as a Arbortext Editor document and displays it in the ActiveX embedded Arbortext Editor window. In contrast to the `open` method, the `show` method enables you to dynamically create an in-memory document that might not have been saved at all and display it in the ActiveX embedded window.

This method first closes the current document (if any) and prompts the user to save it (if modified). Note that at the **Save** prompt, the user might **Cancel** the operation. In this case, the current document is left intact and this method returns an error.

If you do not want the user to see a Save prompt, then you can call the `close` method to close the current document explicitly before calling the `show` method.

`show`(documentId [, windowFlags [, xuiPath]])	
Parameters	`String` *documentId* Represents the ACL document identifier for an already document. `int` *windowFlags* [optional] Defaults to 0. A bitmask that specifies window options. Constructed by ORing the bits from the following enumeration: • `0x00001` - Supply vertical scrollbar (pane). • `0x00002` - Supply menu bar. If this is set then the **Menu** toolbar button will be shown. If this is not set, the

	Menu toolbar button is not shown and there will be no

	way to bring up menus. The menu bar is part of the Edit toolbar (Toolbar 1) — 0x00020. Toolbar 1 must be supplied to enable the display of the menu bar. • `0x00004` - Supply command subwindow. • `0x00008` -Supply message footer subwindow. • `0x00010` - Automatically call `ADocument.close()` on the attached document when the window is destroyed. (pane). • `0x00020` -Supply the Edit toolbar (that is, Toolbar 1) (pane). • `0x00080` - Supply horizontal scrollbar (pane). • `0x00100` - Do `edit` command intializations, include reading the document type instance command files (`instance.acl` and `instance.js`) and document command files (docname`.acl` and docname `.js`) if they exist, and calling the ACL `editfilehook` when a document is attached to the window. This bit applies only to edit class windows (pane). • `0x01000` - Supply a table column width ruler (pane). • `0x02000` - Supply a table row height ruler (pane). • `0x04000` - Supply the Markup toolbar (that is, Toolbar 2). • `0x08000` - Supply the Table toolbar (that is, Toolbar 3).

	• `0x10000` - Supply the Application toolbar (that is, Toolbar 4). If a menu bar is requested, it must be initialized using the **menu_load** or **menu_add** ACL commands before the window is first displayed. If a message footer is created, error messages and output from the **message** ACL command are displayed in the left part of the footer if the message is short enough (otherwise a popup dialog box is used). Any messages directed to the message footer are considered transient and are erased on the next key or button event received in the window. If this parameter is omitted or zero, it behaves as if the following value were given: `0x9F9FF`. This corresponds to the internal ACL constants (*h::winMaskMain* | *h::winMaskEditStyle*) which are used to create default "edit" windows. Over time, these constant might change, so this ensures the latest defaults are used. Regardless of the value given, the following bits will be forcibly set:`0x80850`. This ensures that some window properties are set that are absolutely required for the control to work properly. To cause the document to be shown with an absolute minimum of window artifacts (toolbars, scroll bars, status bar, etc..), any non-empty subset of `0x80850` can be used. For example: `0x00040`. **Note** If a `0` value is given, the `0x9F9FF` value is used. this value includes the `0x00010` bit, which causes the document to be closed when the user closes the Embedded Frame window. If you want the given document to remain open after the Embedded Frame window is closed, then don't use a `0` value for the *windowFlags*. Instead, use a non-zero bitmask that does not have the `0x00010` bit set. `String` *xuiPath* [optional] Defaults to an empty string. An optional parameter used to supply an alternative XUI file to define the toolbars used by the edit window. If *xuiPath* is not supplied (or empty), then *`Arbortext-path`*`\lib\dialogs\editwindow.xml` is used.
Returns	The ACL ID of the possibly new window that was loaded

	as a result of this call. A value of `-1` can be returned if a *documentId* parameter value of `-1` was given.
Throws	A COM error will be returned to the caller for the following cases: • User canceled at the save prompt or at some prompt while opening the requested document. In this case, returns an HRESULT value of `0x800704C7` which enables the caller to distinguish this case from other types of failures. This represents a FACILITY of `FACILITY_WIN32` and a WIN32 error code of `ERROR_CANCELLED`. • An invalid *documentId* was provided. In this case. the HRESULT will be `E_FAIL`

close Method

Closes any open document after first prompting to save any changes (if it was modified).

`close`([, closeFlags])	
Parameters	`int` *closeFlags* [optional] Defaults to 0. A bitmask that specifies close options. Constructed by ORing the bits from the following enumeration: • `0x0001` - When closing the current document, suppress the prompt to save the current document. • `0x0002` - After closing the current document, do not display even the façade document. Note that in this case the host window will show through the ActiveX control area, which might result in unexpected display results. • `0x0004` - When closing the current document and a prompt is put up to save, suppress the option to Cancel. This option is useful if the containing host window is closing in such a way that cannot be aborted or canceled. It enables users to save a document without giving them the impression that they can cancel the operation that is closing down their document.

Returns	None.
Throws	A COM error will be returned to the caller for the following case: • User canceled at the save prompt. In this case, returns an HRESULT value of `0x800704C7` which enables the caller to distinguish this case from other types of failures. This represents a FACILITY of `FACILITY_WIN32` and a WIN32 error code of `ERROR_CANCELLED`. In general, it is very unlikely that this method will fail other than the Cancel case.

Integrating Arbortext Editor with Web Pages

On the Microsoft Windows platform, you can set up links in web pages that will open an Arbortext Editor session.

You can use the `arbortext-editor` protocol (or scheme) in a URI (Uniform Resource Identifier) to invoke Arbortext Editor from a link. Sample web integration implementations are available in the *`Arbortext-path`*`\samples\web-integration` directory. Refer to the `readme.txt` file in that directory for a description of the samples.

The Protocol Syntax

Following is the syntax for the `arbortext-editor` protocol presented in the RFC format defined in RFC 4395 by the Internet Assigned Numbers Authority (IANA):

```
arbortexturi = scheme ":" resource [ "?" query] [ "#" anchor]
scheme = "arbortext-editor" |
query = query-pair *["&" query-pair]
query-pair = key "=" value
resource = utf8 url encoded resource
key = utf8 url encoded query component
value = utf8 url encoded query component
anchor = utf8 url encoded resource
```

In the syntax, `resource` must be the encoded full path to a document that can be opened with Arbortext Editor.

`utf8 url encoded resource` refers to the standard rules for encoding a resource referenced from a URI. The following rules apply in this case:

- The alphanumeric characters "a" through "z", "A" through "Z" and "0" through "9" remain the same.
- The special characters " . ", " - ", " * " and " _ " remain the same.
- The space character is converted into "%20".
- All other characters are first converted into one or more bytes using UTF-8, and then each byte is represented by the character string "%*xy*", where *xy* is the two-digit hexadecimal representation of the byte.

For example, the string would get converted to , because in UTF-8 the character `ü` is encoded as two bytes `C3` (hex) and `BC` (hex), and the character @ is encoded as `%40` and spaces are encoded as `%20`.

`utf8 url encoded query component` refers to the optional query string, which must be encoded according to the `application/x-www-form-urlencoded` mime type (defined in the W3C HTML 4.01 Specification). The basic encoding rules are the same in this case, but space characters are represented by a plus sign (+).

Following are some examples of legal uses of the protocol:

`arbortext-editor:http%3A%2F%2Fserver%2Fpath%2Ffile.xml`	
Protocol	`arbortext-editor`
Unencoded Resource	`http://server/path/file.xml`

`arbortext-editor:c%3A%5Ctemp%20dir%5Cfile%CE%A8.xml?name=Robert+Smith#Terrier`	
Protocol	`arbortext-editor`
Unencoded Resource	`c:\temp dir\fileΨ.xml`
Unencoded Query Name	name
Unencoded Query Value	Robert Smith
Anchor	Terrier

The `arbortext-editor` Protocol

You use the `arbortext-editor` protocol to launch a Arbortext Editor session for a document from a link on a web page. In this case, the usual Arbortext Editor window is used to open the document, so Arbortext Editor must be installed or otherwise available on the local system. In general, when a web browser processes a link with an unknown protocol the browser checks the local system to see

whether there is an application registered to handle the protocol. When Arbortext Editor is registered as a COM server on a Microsoft Windows system, part of that process associates Arbortext Editor with the `arbortext-editor` protocol.

For security reasons, most web browsers usually prompt the user before initially launching the associated program for a given protocol URI. Since Arbortext Editor is registered as the associated program for the `arbortext-editor` protocol, a browser generally displays a dialog box warning the user that this link will launch an application on their system and asking for confirmation before proceeding. Such dialog boxes generally have an option to not show the warning again for this protocol. If Arbortext Editor is not installed on the system, or is not registered for some reason, most browsers will display an error message saying that the protocol is not associated with an installed program.

Following is an example of an HTML link that would open Arbortext Editor for a document named `sample.xml`:

```
<a href="arbortext-editor:http%3A%2F%2Fdocserver%2Fwebdav%2Fsample.xml">
Sample document
</a>
```

Note that if the link appears as an attribute value in HTML markup, then it is subject to the normal encoding rules of those attribute values. In particular, if the link contains the `&` character (such as if there is a query string with more than one parameter), then this must be encoded as `&`. For example, the following link:

```
arbortext-editor:http%3A%2F%2Fdocserver%2Fwebdav%2Fsample.xml?color
=blue&priority=high
```

should appear inside of an HTML link as follows:

```
<a href="arbortext-editor:http%3A%2F%2Fdocserver%2Fwebdav%2Fsample.xml?color
=blue&amp;priority=high">
Sample document
</a>
```

The Security Zone Policy

By default, whether a document is opened by an `arbortext-editor` protocol link is determined by the Microsoft URL security zone policy. When a link using the protocol is invoked from a web browser, Arbortext Editor first determines the zone of the encoded resource path. The link is then allowed or denied according to this policy:

- Local Machine zone — allow
- Local Intranet zone — allow
- CMS zone — allow

  This a new zone not in the default Microsoft URL security zones. If Arbortext Editor determines that the encoded path is the Logical ID for an object stored in a content management system, then it is considered to be in the CMS zone.

Refer to the *Content Management Guide* for more information about Logical IDs.

- Trusted Sites zone — allow
- Internet zone — deny
- Restricted Sites zone — deny

If the link is denied, then a message is displayed saying that the request has been denied for security purposes. The use of URL security zones is controlled by the `webzonepolicy` preference.

Processing Query Strings

The `arbortext-editor` protocol enable you to add an optional query string to the URI. Following is an example of a URI with the query string:

```
arbortext-editor:x-wc%3A%2F%2Ffile%3D1234.xml?workspace=ws1&hosturl=
http%3A%2F%2Fpjl%2FWindchill
```

In this example, there are two query parameters: `workspace` and `hosturl`. Their decoded values are `ws1` and `http://pjl/Windchill`.

These query string parameters enable the link to specify potentially useful metadata about the link. This metadata can be accessed inside of an ACL `editbeforehook` hook function. When one of these links are selected in a web browser, all of the functions in the `editbeforehook` are called before the document is actually opened. This gives the hook function a chance to do any special processing before the document opens.

For example, following is an HTML link containing some query strings that opens an Arbortext Editor session:

```
<a href="arbortext-editor:http%3A%2F%2Fserver%2Fpath%2Ffile.xml?
hint1=food&hint2=M%26Ms">
click here
</a>
```

This links opens the `http://server/path/file.xml` document and includes the hints `food` and `M&Ms` as query strings. Assume the following hook code is sourced:

```
package metadata;

function my_edit_before_hook(path)
{
 local hint1 = get_custom_property('com.ptc.arbortext.launcher.temp.hint1');
 local hint2 = get_custom_property('com.ptc.arbortext.launcher.temp.hint2');

 response("Path=$path\n\nhint1=$hint1\n\nhint2=$hint2");

 # returning -1 will cancel the edit
}
```

```
add_hook('editbeforehook', package_name().'::my_edit_before_hook');
```

In this case, when the link is selected in the web browser, the response dialog box displays both the decoded `http` path to the file and the `hint1` and `hint2` metadata that was in the original HTML link.

Accessing Query String Parameters

While the `editbeforehook` functions are being called, the query strings are available through the following function and methods:

- ACL: `get_custom_property()`
- AOM: `Application.getCustomProperties().getString()`

After the hook functions have all been called, the query string values are no longer available. To access a query string parameter, you prefix the parameter name with the following value before calling `get_custom_property()` or `Application.getCustomProperties().getString()`: `com.ptc.arbortext.launcher.temp.`

For example, to access the `workspace` parameter in the following example link:
`arbortext-editor:x-wc%3A%2F%2Ffile%3D1234.xml?workspace=ws1&hosturl=http%3A%2F%2Fpjl%2FWindchill`

You could use the following ACL code:

```
local value = get_custom_property('com.ptc.arbortext.launcher.temp.workspace');
```

You could also use the following AOM code:

```
String value = Application.getCustomProperties().getString("com.ptc.arbortext.launcher.temp.workspace");
```

If there is no such query parameter, then an empty string is returned for ACL and a `null` is returned for AOM.

Note

The values returned are the fully decoded string values. If a link contained the following query string parameter:

`hosturl=http%3A%2F%2Fpjl%2FWindchill`

The following value would be returned:

`http://pjl/Windchill`

8

Working with Arbortext Import/ Export

Note

Although still a part of the Arbortext technology stack, this feature has been deprecated and is no longer supported by PTC.

Configuring for Exporting

Export Process Overview

Arbortext Import/Export uses Arbortext Styler to export XML documents to RTF. To install and configure Arbortext Import/Export for Export stylesheet development and deployment, perform the following steps:

1. Ensure that Arbortext Import/Export has been activated on client workstations and the Arbortext Publishing Engine server following the instructions in the *Installation Guide for Arbortext Editor, Arbortext Styler, and Arbortext Architect* and *Installation Guide for Arbortext Publishing Engine*.
2. Configure your workstation for creating Export Stylesheets. Refer to Creating Export Stylesheets on page 246
3. Create Export stylesheets using Arbortext Styler. Detailed instructions on creating Export stylesheets are available in the Exporting XML Documents to RTF section of the Arbortext Styler online help available from the Arbortext Styler **Help** menu.
4. To deploy completed Export stylesheets for use from client workstations, copy the stylesheets to the Arbortext Publishing Engine server. Refer to Deploying Export Stylesheets on page 247.
5. Optionally, configure RTF export settings for the users at your site using the Export stylesheets to export documents to RTF. Refer to Configuring Client Workstations on page 247.

Arbortext Import/Export exports documents as RTF version 1.7 documents. Visit www.microsoft.com for a copy of the Rich Text Format (RTF) Specification Version 1.7.

Creating Export Stylesheets

Export stylesheets are created using Arbortext Styler. Use the RTF property category in Arbortext Styler to specify the style characteristics for each XML structure in the document. Refer to *Publishing XML Documents as RTF Files* in the Arbortext Styler online help for detailed information on creating Export stylesheets.

Deploying Export Stylesheets

As with all stylesheets, Export stylesheets are made available to client workstations from the Arbortext Publishing Engine server. Once your Export stylesheet(s) have been created and tested, copy them to your Arbortext Publishing Engine server's `\custom\importexport` directory and restart your Arbortext Publishing Engine server.

Refer to Overview of Custom Programs and Scripts on page 12 for details on working with the `\custom` directory.

Configuring Client Workstations

Configure client workstations in the following manners as appropriate.

Customize RTF Style and Field Names

The paragraph styles, character styles, list paragraph styles, and field names available in Arbortext Styler's RTF property category are defined in two configuration files in the following directory:
Arbortext-path`\lib\importexport\lib`

These files are well-formed, free-form XML files that can used as the basis for custom lists of names. The files are:

- `word-builtin-styles.xml` — Specifies the available paragraph, list paragraph, and character style names in the **RTF style name** and **RTF list name** lists in the **RTF** property category and the **Paragraph Style for Lists** dialog box. By default, the names in the list are the default style names in Word's `normal.dot` template file.
- `word-fields.xml` — Defines the structure of all supported Word fields, along with their instructions, switches, and short descriptions of each field. The specified information is displayed on the **Field** dialog box available from the **RTF** property category.

If you wish to customize the list of names available at your site, PTC recommends that you copy these files to the `\importexport` subdirectory of a custom directory, edit the copies, and set each client workstation to use that custom directory. Arbortext Import/Export will search `\importexport` (and all paths defined by the `importexportpath` Advanced Preference) for the first-found instance of `word-builtin-styles.xml` and `word-fields.xml`.

Associate a Program for Previewing RTF Documents

To preview exported RTF documents from Arbortext Styler, a Windows application must be associated with the `.rtf` file extension. PTC recommends you associate Microsoft Word 97 (or higher) to the `.rtf` file extension. Wordpad is delivered with Windows, and will also open RTF files. However, Wordpad may not support all of the features in the RTF file.

Refer to the Microsoft Windows online help for information on associating a file with a program.

Specifying a Custom Template File

If you want users to use a Word template file other than that shipped with Arbortext Import/Export, create the template file, save the template file from Word in RTF format, and copy the RTF template to the Arbortext Publishing Engine server. From each client workstation, run Arbortext Styler, choose **File** ▸ **Stylesheet Properties**, and navigate to the **RTF** tab. Browse to or enter the path and file name of the RTF template in the **Template File** field. Users may override this setting by changing or deleting the value in this field.

If no template file is specified, only the Word styles defined in *`Arbortext-path`*`\lib\importexport\lib\word-builtin-styles.rtf` will be used for RTF formatting.

Note

Be aware of the following items:

- During stylesheet development using Arbortext Styler, this file name is a fully-qualified path name, which may not be valid on the Arbortext Publishing Engine server when the stylesheet is deployed. In this case, the Arbortext Publishing Engine will search the value of the `importexportpath` Advanced Preference for the first-found instance of the user-defined RTF template file, using the base name of the file.
- Arbortext Import uses an RTF version of the Word template file . If a `.dot` template file is updated, the `.dot` file must be re-saved as a `.rtf` file.

If you wish to include the user-defined style names in the style name lists of the **RTF** property category, you can create a custom version of `word-builtin-styles.xml`. To create the desired XML markup for `word-builtin-styles.xml`, consider using the following VBA macro to extract style names from any Word document or template and create the desired markup for paragraph and character styles. You will then need to extract the list paragraph styles from the `<ParagraphStyles>` list and add `level` attribute to indicate nesting level.

```
Sub PrintAllBuiltinStyles()
'
' Loop through the builtin styles
' and print the rtf-fields.xml markup
' to the "Immediate Window"
'
'
Dim aStyle As Style
Dim aStyles As Styles

Set aStyles = ActiveDocument.Styles
Debug.Print "<ParagraphStyles>"
For Each aStyle In aStyles
 If aStyle.BuiltIn = True Then
 If aStyle.Type = wdStyleTypeParagraph Then
 Debug.Print "<StyleName>" + aStyle.NameLocal + "</StyleName>"
 End If
 End If

Next
Debug.Print "</ParagraphStyles>"
Debug.Print "<CharacterStyles>"
For Each aStyle In aStyles
 If aStyle.BuiltIn = True Then
 If aStyle.Type = wdStyleTypeCharacter Then
 Debug.Print "<StyleName>" + aStyle.NameLocal + "</StyleName>"
 End If
 End If

Next
Debug.Print "</CharacterStyles>"

End Sub
Sub PrintAllUserDefinedStyles()
'
' Loop through the builtin styles
' and print the rtf-fields.xml markup
' to the "Immediate Window"
'
'
Dim aStyle As Style
Dim aStyles As Styles

Set aStyles = ActiveDocument.Styles
Debug.Print "<ParagraphStyles>"
For Each aStyle In aStyles
 If aStyle.BuiltIn = False Then
 If aStyle.Type = wdStyleTypeParagraph Then
 Debug.Print "<StyleName>" + aStyle.NameLocal + "</StyleName>"
 End If
 End If
```

```
Next
Debug.Print "</ParagraphStyles>"
Debug.Print "<CharacterStyles>"
For Each aStyle In aStyles
 If aStyle.BuiltIn = False Then
 If aStyle.Type = wdStyleTypeCharacter Then
 Debug.Print "<StyleName>" + aStyle.NameLocal + "</StyleName>"
 End If
 End If

Next
Debug.Print "</CharacterStyles>"

End Sub
```

Configuring for Importing

Configuring Client Workstations

Configure client workstations for importing files in the following manners as appropriate.

Using Arbortext Import/Export in Batch Mode

You can use Arbortext Import/Export to import and export documents in an unattended, or batch, fashion. From the Arbortext Editor command line or an ACL script, use the following ACL functions when importing and exporting documents in batch mode.

document_export ([*inFile*[, *outFile*[, *styleSheet*[, *logFile*[, *nFlags*]]]]])

Log Files, Error Return Codes, and Event Log Errors

The Arbortext Import/Export ACL functions and general import, export, and mapping functionality use the event log mechanism. Error return codes for these operations are defined in the following table.

Import and Export Return Codes

Return code	Description
-1	Platform is not MS Windows.
0	No error.
1	No *HOME* directory defined.
2	No file selected.
3	Arbortext Import/Export feature not installed.
4	Specified Repository directory is in the install tree.
5	Specified Repository directory not found.
6	Specified Repository directory is missing required subdirectories.
7	Cannot open `\importexport\config\XYZ_SysPrefs.xml`.
8	Cannot create specified Arbortext Import Workbench path.
9	Cannot create specified Arbortext Import configuration path.
10	Cannot copy default configuration from install tree.
11	Cannot create Repository directory.
12	Attempt to create project in the install tree.
13	Arbortext Import project fatal exception.
14	Invalid Import options.
15	File copy operation failed.
16	Directory copy operation failed.
17	Repository copy operation failed.
18	Project copy operation failed.
19	Operation cancelled by user.
20	Project .exlst not found.
21	Project .exlst is invalid.
22	No Arbortext Import Workbench license found.
23	No Arbortext Import/Export license found.
24	Cannot create backup copy.
25	Undefined error.
26	Path too long.
27	Cannot open driver factory.
28	Configuration directory does not exist.
29	`sysprefs` path not found.
30	Configuration directory is network path.

Import and Export Return Codes (continued)

Return code	Description
31	Configuration directory copied.
32	Cannot load conversion bridge DLL.
33	Migration no longer supported.

Troubleshooting

Platform Issues

If you receive the following error message:
`Failed to load conversion bridge DLL`

.NET Framework v2.0.50727 or v3.0 may not be installed. Download an updated .NET Framework from www.microsoft.com.

If you receive the same error message with return code 30 and a path starting with \\, this indicates that the executable is on a network drive. This configuration is not supported due to .Net library restrictions.

Improving Import Performance and Freeing Disk Space

During each import, temporary directories and files are created and stored in the following directory in the Project Directory:
`\Sample\ExecutionResults`

Manually delete the contents of this directory over time to increase performance and free disk space. Do not delete the directory if it's still used with an active project.

Importing Very Large Documents

When attempting to import very large documents (such as 50MB+ MIF files that publish as several-hundred-page documents), Arbortext Publishing Engine may report exception errors and fail to import the documents. In these situations, consider trying the following workarounds to successfully import the documents.

- Break the large document into several smaller documents and import the smaller documents.
- Raise the maximum size of the Java Virtual Machine (JVM) memory allocation pool by giving a higher value to one of these settings:

- APTJAVAVMMEMORY environment variable — sets the size of the pool at startup of Arbortext Editor, when the default JVM is loaded
- `javavmmemory` set option and Advanced Preference — sets the size of the pool at any point before the JVM starts

> **Note**
> If APTJAVAVMMEMORY has a value, any `set javavmemory` commands are ignored.

Be aware that setting this value to too high of a value will result in a failure to start the Java Virtual Machine. If changing this value results in the document being successfully imported, be sure to reset the `javavmmemory` value. Having this value unnecessarily high may result in other publishing and importing issues.

Known Import Limitations

Be aware of the following limitations when using Arbortext Import/Export to import documents:

- Import to SGML is not directly supported by Arbortext Import/Export. To import SGML files, create an XML version of the target document type and use an ACL `postimport` hook to save as SGML
- Arbortext Import/Export cannot import graphics as entity references.
- Processing instructions cannot be directly imported.
- MIF embedded graphics created using the FrameMaker drawing tools are not imported.
- In rare circumstances, certain embedded images in MIF files cannot be converted from their original format, instead appearing in the imported file as embedded Microsoft Word documents. The embedded images are identified in the FrameMaker MIF files as "OLE2" images, which is the same OLE compound document format as Microsoft Word files. A possible source for this scenario may occur when the images were originally created in Microsoft Word and then pasted into the FrameMaker document. These embedded images must be recreated in a different format before conversion or be manually recreated after the conversion.
- Imported documents are not guaranteed to be contextually valid documents.
- Arbortext Import has no concept of an XML template, which uses Arbortext Editor's caret location as the starting point. This means any boilerplate data must be encapsulated in one or more MapObjects.

- Arbortext Import imports RTF versions 1.7 and 1.8 (although specific features of any version may not be mappable or importable). The hundreds of third-party applications that produce RTF may create RTF files with unexpected content. Arbortext Import may not be able to import all of these variations of RTF.
- Microsoft Word text boxes are not fully supported and may import in unexpected manners due to the nature of their anchor point/XY location relations.
    - Top-down order may vary from the apparent document order as anchor points are often hidden and unknown.
    - Layers of text boxes can obscure each other and can produce unexpected results.
    - Certain fonts may be processed incorrectly.
    - Freeform mixtures of text and images within text boxes, shapes, and groups may result in lost data or out-of-order data.
- Arbortext Import can only process system files. Ensure that permissions are set properly if a database repository is involved in your import process.
- Arbortext Import does not support importing Hebrew, Arabic, and Thai content.

Known Export Limitations

Refer to the Export section of the Arbortext Styler documentation for a listing of Export limitations.

9

Customizing Copying and Pasting from Other Applications

Customizing Copying and Pasting from Other Applications

On the Windows platform, Arbortext Editor enables you to copy content from other applications and paste the content into your document using matching markup from your document type. You can copy content from Microsoft Word, Adobe FrameMaker, web browsers, text editors, and other applications and paste that content into your document. Arbortext Editor uses technology to map the content from the other application to the markup appropriate for your document. For example, assume you are authoring a DITA document and copy a bullet list in a Microsoft Word document. When you paste that list into your DITA document, it will be pasted with the appropriate `ul` and `li` tags.

This chapter provides an overview of this feature and its limitations. It tells you how to disable the feature and how to modify the supported clipboard source types. It also tells you how to implement the feature for custom document types, and how to customize the **Paste Special** dialog box.

Copy and Paste Overview

When you copy text to the Microsoft Windows clipboard, certain applications copy more than just text to the system clipboard. For example, most Microsoft applications put the following formats on the clipboard for a copy operation:

- RTF markup (Rich Text Format)
- HTML markup
- Unicode text
- ANSI text

Adobe FrameMaker puts FrameMaker Interchange Format (MIF) on the clipboard during a copy operation, as well as RTF markup and Unicode and ANSI text.

Using Arbortext Import/Export technology, Arbortext Editor can automatically map content on the clipboard to tagging appropriate for the document type of the document where the content is being pasted. Arbortext Editor generates a basic map template based on the following information:

- The source type of the copied text on the clipboard

  The more formatting information that is provided in the source text, the better the paste operation can map to the document type elements. For example, if just ANSI text is available on the clipboard then all that could be determined is paragraph boundaries. However, if RTF is available on the clipboard then much more information is available to the paste operation. In this case, the paste operation can determine not only paragraphs, but also titles, images, links, tables, and so forth.

Note that some source types explicitly define document elements that can be mapped to markup without much ambiguity. In other cases, the source type implies parts of the document structure that might not reflect the desired result in the converted markup.

The type of document elements that are explicitly defined in a source type vary based on the type. The following table summarizes the document elements defined in the various source types.

Document Element	RTF	MIF	HTML	Text
Paragraphs	X	X	X	X
Titles			X	
Divisions			X	
Tables	X	X		
Images	X		X	
Links and cross references	X	X	X	
Footnotes	X	X		
Index terms	X	X		
Inline emphasis (bold, italic, or underline)	X	X	X	
Link targets (IDs or bookmarks)	X	X	X	
Division titles (H1, H2, and so forth)			X	
Divisions and Titles	X	X		

Divisions and division titles must be inferred in most source types by the use of explicit style names. HTML markup, in contrast to word processor documents, contains actual hierarchical elements that can explicitly define divisions and their titles.

- The document type of the document currently being edited in Arbortext Editor where the text is being pasted
- The document type configuration file (`.dcf`) associated with the document type

For the best results for a paste operation, details about the roles of various tags in the document type must be well defined in the associated `.dcf` file.

- The Arbortext Styler stylesheet (`.style`) associated with the document type

Support for copying and pasting from other applications has been added to the `.dcf` files for the following document types distributed with Arbortext Editor:

- ATI XML DocBook V4.0 (`axdocbook`)
- Arbortext Article (`asdocbook`)
- The DITA Topic and Concept document types
- The Technical Information Application topic document types
- HTML

In most cases, copying and pasting works for both XML and SGML document types because Arbortext Editor has automatic features to handle XML-specific markup in a manner compatible with SGML markup.

While Arbortext Editor automatically creates Import MapTemplate files for these document types, you can use Arbortext Import to create custom maps for your site. For example, your site might use custom Microsoft Word style names site that need to be defined in the MapTemplate, or you might want to define additional inline elements for a paste operation besides the emphasis elements.

If you have an custom document type, you can also configure your document type to intelligently paste content from other applications into Arbortext Editor.

Disabling Copy and Paste

If you do not want Arbortext Editor to convert content from other applications into markup, then you can disable this feature. In this case, copy and paste from other applications will function as that operation did before Arbortext Editor release 5.4. That is, pasted content will be a stream of text without any tagging.

Following are some reasons you might want to revert to the earlier way Arbortext Editor did copy and paste operations from other applications:

- Many applications that want to preserve text formatting in clipboard content use RTF to store that content. This might cause unexpected results in the converted markup.
- This feature is dependent on a certain release of the Java Virtual Machine (JVM) and third-party Java classes. Changes in the `.jar` files in your install tree or in a `custom\classes` directory might adversely affect the copy and paste feature requiring you to disable it.
- If you have existing customizations that process HTML or RTF clipboard content or if you have a customization using the `buffer_clipboard_`

`contents` function, then you might want to retain that customization instead of using the new copy and paste feature.

- If you use a right-to-left language then you might want disable this feature, as Arbortext Import/Export does not reliably support language directionality.

Follow these steps to disable copy and paste:

1. In Arbortext Editor. select **Tools** ▸ **Preferences**.

   The **Preferences** dialog box opens.

2. Click the **Advanced** button.

   The **Advanced Preferences** dialog box opens.

3. Select the **pastesource** preference and click the **Edit** button.
4. Delete all of the content in the **Value** field and click **OK**.
5. Click **Close** to close the **Advanced Preferences** dialog box and **OK** to close the **Preferences** dialog box.

   Copy and paste is now disabled.

Note that you can also use the `set pastesource` command to disable copy and paste by setting the value of the command to an empty or null string, for example: `set pastesource=""`.

Modifying the Source Types Used for Copy and Paste

In addition to disabling copy and paste, you can also control the clipboard formats Arbortext Editor uses to convert the copied content to markup. Following are some reasons you might want to control the clipboard formats:

- Adobe FrameMaker puts both MIF and RTF content on the clipboard during a copy operation. You might want to remove the MIF format so that RTF is used for converted markup.
- Many applications put both HTML and RTF content on the clipboard during a copy operation. You might want to disable one or the other of these formats for a particular application because you get better results from a specific format.
- You might want to disable all of the formats except for text, but still enable users to use **Edit** ▸ **Paste Special** and the **Paste Special** dialog box to provide some control over the results of a paste operation.

Follow these steps to control the source types used for copy and paste:

1. In Arbortext Editor. select **Tools** ▸ **Preferences**.

The **Preferences** dialog box opens.

2. Click the **Advanced** button.

   The **Advanced Preferences** dialog box opens.

3. Select the **pastesource** preference and click the **Edit** button.
4. Set the content in the **Value** field to the clipboard format(s) you want to use and click **OK**.

   The following values are supported:

   - `htm` — HTML markup
   - `mif` — Maker Interchange Format (MIF), the document format supported by Adobe FrameMaker
   - `rtf` — Rich Text Format (RTF), the document format used by several Microsoft applications including Microsoft Word
   - `txt` — Unicode and 8-bit ANSI text

   Multiple values must be separated by semnicolons (;).

5. Click **Close** to close the **Advanced Preferences** dialog box and **OK** to close the **Preferences** dialog box.

Note that you can also use the `set pastesource` command to set the clipboard formats supported for copy and paste.

Using Arbortext Import to Customize the MapTemplate Files

You can use the Arbortext Import MapTemplate Editor included in Arbortext Architect to modify the automatically generated MapTemplate files that Arbortext Editor produces for a copy and paste operation. Following are some reasons you might want to modify the default templates:

- Modify the division title MapObjects to match custom heading style names instead of Microsoft Words default “Heading *n*” style names.
- Modify the behavior of the default ID/IDREF handling from Microsoft Word bookmarks or Adobe FrameMaker cross-reference markers.
- Add new MapObjects to convert inline character styles to elements
- Add new MapObjects to convert formatted text to both elements and attributes
- Modify the default MapObjects to handle user-defined list paragraph styles, list styles, or tables styles
- Add map debug comments to troubleshoot copy and paste conversion issues
- Add new MapObjects to work with custom **Paste Special** operations

The automatically generated MapTemplate files

The first time that you copy and paste content from another application into Arbortext Editor, a MapTemplate file for the pasted clipboard content and the document type into which the content is pasted is automatically created. These files are stored in the Arbortext Editor cache directory (`.aptcache`). The cache directory is typically located at `C:\Users\<username>\Application Data\PTC\Arbortext\Editor\.aptcache`. The MapTemplate files are stored in a subdirectory of `.aptcache` named `maptemplates`.

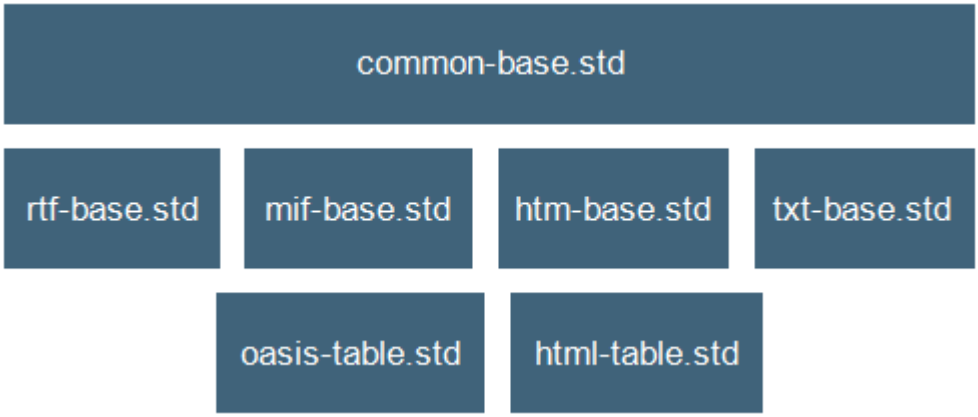
The MapTemplate files are named *`source_type-doctype_name`*`.std`. For example, a MapTemplate file for the RTF source type and the axdocbook document type would be named `rtf-axdocbook.std`. A file for the HTML source type and the DITA topic document type would be named `html-topic.std`.

You can modify these automatically generated files in the MapTemplate Editor as required for your site. Put the modified files in either the *`Arbortext-path`*`\custom\lib` directory or in a `custom\lib` directory referenced through the *APTCUSTOM* environment variable. Your customized MapTemplate file will override the automatically generated file for the source type and document type combination implied by its file name.

The automatically generated MapTemplate files are regenerated every time you start a new Arbortext Editor session and paste for the first time with a given source type and document type combination. If you modify a document type's `.dcf` file or perform a Arbortext Editor preview operation in Arbortext Styler, the MapTemplate files in `.aptcache\maptemplates` are marked as outdated and are regenerated as if you have started a new Arbortext Editor session. MapTemplate files in a `custom\lib` directory are not affected by Arbortext Editor operations and can only be disabled by renaming the file or deleting the MapTemplate from the file system.

The base MapTemplate files

The automatically generated MapTemplate files are created from a set of base MapTemplate files. These files are stored in the *`Arbortext-path`*`\lib\cpix\lib` directory. The following base MapTemplate files are in this directory:



Base MapTemplate files

- `common-base.std` — The common file used for all automatically generated maps.
- `htm—base.std` — The file used for HTML content.

  This file is concatenated with the common file to generate an HTML MapTemplate file. Similarly, one of the following three base files is concatenated with the common file to create MapTemplate files for the associated type of content. Only one of these four files is used for an automatically generated MapTemplate file.
- `mif—base.std` — The file used for MIF content.
- `rtf—base.std` — The file used for RTF content.
- `txt—base.std` — The file used for text content.
- `html-table.std` — The file used for HTML tables.

  This file is concatenated with the common file and the source type file to create the final MapTemplate file. The type of table file used is based on the primary table model supported by the document type. Only one of the table files is included in an automatically generated MapTemplate file.
- `oasis-table.std` — The file used for OASIS Exchange tables.

The base MapTemplate files contain placeholder values for the document type element names. For example, the element used for paragraphs in a document type has the placeholder `aticp:primary_paragraph` in the Output Rules defined in the base MapTemplate file. This placeholder is replaced in the automatically generated MapTemplate file by the paragraph element defined in the `.dcf` file for the document type used for the generated file. Other elements are similarly added to the file based on information in the document type's `.dcf` and `.style` files.

Following are the placeholder values for different document elements in the base MapTemplate files, including example markup from the `axdocbook.dcf` file that matches the placeholders for that document type.

aticp:primary_paragraph	
Description	Primary paragraph element

Sample Markup	`<Specials>`
	`<Paragraph`
	`element="para"/>`
	`</Specials>`

Notes	You can identify the element with this ACL function: `paragraph_tag_name(`*`doc`*`)`

aticp:division1	
Description	Primary division element

Sample Markup	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect1"`
	`primary="yes"/>`
	`</ElementOptions>`

Notes	You can identify division elements with this ACL function: `division_tag(`*`tagname`*`[, `*`doc`*`[, `*`primary`*`]]))`

aticp:division2	
Description	Nested division element

Sample Markup	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect2"`
	`primary="yes"/>`
	`</ElementOptions>`

Notes	

aticp:division3	
Description	Nested division element

Sample Markup	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect3"`
	`primary="yes"/>`
	`</ElementOptions>`

Notes	

aticp:division4	
Description	Nested division element

Sample Markup	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect4"`
	`primary="yes"/>`
	`</ElementOptions>`

Notes	

aticp:division5	
Description	Nested division element

Sample Markup	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect5"`
	`primary="yes"/>`
	`</ElementOptions>`

Notes	

aticp:divisiontitle	
Description	Primary division title element

Sample Markup	`<PasteOptions>`
	`<PasteElement`
	`category="primary_division_title"`
	`element="title"/>`
	`</PasteOptions>`

Notes	You can identify the element with this ACL function: `dcfmodel_element_list(`*`arr`*`, primary_division_title[, `*`doc`*`[, 1]])`

aticp:bold	
Description	Markup for bold text
Sample Markup	`<TextStyles> <Bold attribute="role" attributeValue="bold" element="emphasis"/> </TextStyles>`
Notes	You can identify the element with this ACL function: `text_style_tag_name(bold, `*`arr`*`[, `*`doc`*`])`

aticp:italic	
Description	Markup for italic text
Sample Markup	<TextStyles> <Italic attribute="role" attributeValue="italic" element="emphasis"/> </TextStyles>
Notes	You can identify the element with this ACL function: text_style_tag_name(italic, *arr*[, *doc*])

aticp:underline	
Description	Markup for underline text
Sample Markup	<TextStyles> <Underline attribute="role" attributeValue="underline" element="emphasis"/> </TextStyles>
Notes	You can identify the element with this ACL function: text_style_tag_name(underline, *arr*[, *doc*])

aticp:smallcaps	
Description	Markup for smallcaps text
Sample Markup	<TextStyles> <SmallCaps attribute="role" attributeValue="smallcaps" element="emphasis"/> </TextStyles>
Notes	You can identify the element with this ACL function: text_style_tag_name(smallcaps, *arr*[, *doc*])

aticp:subscript	
Description	Markup for subscript text
Sample Markup	<TextStyles> <Subscript element="subscript"/> </TextStyles>
Notes	You can identify the element with this ACL function: text_style_tag_name(subscript, *arr*[, *doc*])

aticp:superscript	
Description	Markup for superscript text

Sample Markup	`<TextStyles>`
	`<Superscript`
	`element="superscript"/>`
	`</TextStyles>`

Notes	You can identify the element with this ACL function: `text_style_tag_name(superscript, arr[, doc])`

aticp:bulleted_list	
Description	Markup for bulleted list

Sample Markup	`<Lists>`
	`<Bulleted>`
	`<Block element="itemizedlist"/>`
	`<Item element="listitem"/>`
	`</Bulleted>`
	`</Lists>`

Notes	This placeholder triggers the main element name and all other list block tags in which this list might be nested.

aticp:bulleted_list_item	
Description	Markup for bulleted list item

Sample Markup	`<Lists>`
	`<Bulleted>`
	`...`
	`<Item element="listitem"/>`
	`</Bulleted>`
	`</Lists>`

Notes	

aticp:numbered_list	
Description	Markup for numbered list

Sample Markup	`<Lists>`
	`<Numbered>`
	`<Block element="orderedlist"/>`
	`<Item element="listitem"/>`
	`</Numbered>`
	`</Lists>`

Notes	This placeholder triggers the main element name and all other list block tags in which this list might be nested.

aticp:numbered_list_item	
Description	Markup for numbered list item

Sample Markup	`<Lists>`
	`<Numbered>`
	`...`
	`<Item element="listitem"/>`
	`</Numbered>`
	`</Lists>`

Notes	

aticp:simple_bulleted_list	
Description	Simplified markup for a bulleted list

Sample Markup	`<Lists>`
	`<Bulleted>`
	`<Block element="itemizedlist"/>`
	`<Item element="listitem"/>`
	`</Bulleted>`
	`</Lists>`

Notes	This placeholder makes no provision for nested lists. It is only used for the **bullet list** selection in the **Paste Special** dialog box.

aticp:simple_numbered_list	
Description	Simplified markup for a numbered list

Sample Markup	`<Lists>`
	`<Numbered>`
	`<Block element="orderedlist"/>`
	`<Item element="listitem"/>`
	`</Numbered>`
	`</Lists>`

Notes	This placeholder makes no provision for nested lists. It is only used for the **numbered list** selection in the **Paste Special** dialog box.

aticp:internal_link	
Description	Markup for internal document links

Sample Markup	`<Specials>`
	`<Link`
	`element="link"`
	`idref="linkend"/>`
	`</Specials>`

Notes	For non-DITA document types, you can identify the element with this ACL function: `link_tag_name([`*`doc`*`])`. For DITA document types, use the `_cpix::make_internal_link(`*`doc`*`)` and `_cpix::get_link_element()` functions. These functions can be customized and do not use the `.dcf` file.

aticp:internal_link_attribute	
Description	The reference attribute for internal document links

Sample Markup	`<Specials>`
	`<Link`
	`element="link"`
	`idref="linkend"/>`
	`</Specials>`

Notes	For non-DITA document types, you can identify the element with this ACL function: `link_idref_attr_name(`*`tagname`*`[,`*`doc`*`])`. For DITA document types, use the `_cpix::make_internal_link(`*`doc`*`)` and `_cpix::get_link_attribute()` functions. These functions can be customized and do not use the `.dcf` file.

aticp:external_link	
Description	Markup for external links
Sample Markup	`<Specials>` `<Link element="ulink" uri="url"/>` `</Specials>`
Notes	For non-DITA document types, you can identify the element with this ACL function: `link_tag_name([`*`doc`*`])`. For DITA document types, use the `_cpix::make_external_link(`*`doc`*`)` and `_cpix::get_link_element()` functions. These functions can be customized and do not use the `.dcf` file.

aticp:external_link_attribute	
Description	The reference attribute for external links

Sample Markup	`<Specials>`
	`<Link`
	`element="ulink"`
	`uri="url"/>`
	`</Specials>`

Notes	For non-DITA document types, you can identify the element with this ACL function: `link_idref_attr_name(`*`tagname`*`[,`*`doc`*`])`. For DITA document types, use the `_cpix::make_external_link(`*`doc`*`)` and `_cpix::get_link_attribute()` functions. These functions can be customized and do not use the `.dcf` file.

aticp:graphic	
Description	Markup for block graphics
Sample Markup	<Specials> <Graphic element="graphic" ... /> </Specials>
Notes	You can identify the element with this ACL function: `graphic_tag_name([doc[, prompt]])`

aticp:graphic_height_attribute	
Description	The height attribute for block graphics
Sample Markup	<Specials> <Graphic ... reproDepth="depth" ... /> </Specials>
Notes	You can identify the element with this ACL function: `graphic_attr_name(tagname, repodep[, doc])`

aticp:graphic_width_attribute	
Description	The width attribute for block graphics
Sample Markup	<Specials> <Graphic ... reproWidth="width" ... /> </Specials>
Notes	You can identify the element with this ACL function: `graphic_attr_name(tagname, repowid[, doc])`

aticp:graphic_scalefit_attribute_value	
Description	The scalefit attribute for block graphics
Sample Markup	<Specials> <Graphic ... scaleToFit="scalefit" ... /> </Specials>
Notes	You can identify the element with this ACL function: `graphic_attr_name(tagname, scalefit[, doc])`

aticp:inline_graphic	
Description	Markup for inline graphics

Sample Markup	`<Specials>`
	`<Graphic`
	`element="inlinegraphic"`
	`... />`
	`</Specials>`

Notes	You can identify the element through the first inline graphic element returned with this ACL function: `dcfmodel_element_list(`*`arr`*`, 'graphic', [`*`doc`*`], 0)`.

aticp:inline_graphic_height_attribute	
Description	The height attribute for inline graphics
Sample Markup	Same attribute as block graphics
Notes	Same ACL function as block graphics

aticp:inline_graphic_width_attribute	
Description	The width attribute for inline graphics
Sample Markup	Same attribute as block graphics
Notes	Same ACL function as block graphics

aticp:inline_graphic_scalefit_attribute_value	
Description	The scalefit attribute for inline graphics
Sample Markup	Same attribute as block graphics
Notes	Same ACL function as block graphics

aticp:default_table_with_title	
Description	Markup for the document type's default table model when a title element is required

Sample Markup	`<PasteOptions>`
	`<PasteElement`
	`category="primary_table_wrapper"`
	`element="informaltable"/>`
	`</PasteOptions>`

Notes	If the document type contains only one supported table model, then that is the default model. Otherwise, you can define the default table model in the `.dcf` file.

aticp:default_table_without_title	
Description	Markup for the document type's default table model when there is no title element
Sample Markup	Same as above
Notes	Same as above

aticp:table_title	
Description	Markup for the table title
Sample Markup	None
Notes	This element is defined by the document type and the content model of the table markup. You can identify the element with this ACL function: `tbl_model_table_title(`*`tmid`*`)`

aticp:table_with_title	
Description	Markup for the table wrapper element when a title element is required
Sample Markup	None required
Notes	This is the element defined by the *primary_table_wrapper* category in the `.dcf` file's `PasteElement` element paste option. If this is not defined in the `.dcf` file and there is no default table model, then this is the first table in the document type's table model list that has a required title element.

aticp:table_without_title	
Description	Markup for the table wrapper element when a title element is not required
Sample Markup	None required
Notes	This is the element defined by the *primary_table_wrapper* category in the `.dcf` file's `PasteElement` element paste option. If this is not defined in the `.dcf` file and there is no default table model, then this is the first table in the document type's table model list that has a optional or no title element.

aticp:source_type	
Description	Source type for the clipboard data
Sample Markup	None
Notes	This placeholder is defined by the type of copy operation taking place based on the defined precedence of the clipboard data. This is used to support the **Paste Special** feature by helping define the correct source type for that operation.

aticp_graphic_attr_name	
Description	The reference attribute for graphics

Sample Markup	Possibly set through:
	`<Specials>`
	`<Graphic`
	`...`
	`entity="entityref"`
	`filename="fileref"`
	`... />`
	`</Specials>`

Notes	If this is not in the `.dcf` file, it is determined from the document type. This is set through the ACL function `oid_set_graphic_pathname()`. This placeholder is used internally and dynamically in place of hard coded attribute names. This makes the MapObject more generic. This operation is performed in an ACL paste callback named `checksmartcopypaste` which is located in *`Arbortext-path`*`\packages\main\_cpix.acl`.

aticp_id_attr_name	
Description	The ID attribute

Sample Markup	Possibly set through:
	`<Options`
	`...`
	`idAttribute="id"`
	`... >`
	`</Options>`

Notes	You can identify the element with this ACL function: `target_id_attr_name(`*`tagname`*`[, `*`doc`*`])`. For DITA document types, this is defined in the `.dcf` file. For other document types, it is defined in the document type. This placeholder is used internally and dynamically in place of hard coded attribute names. This makes the MapObject more generic.

aticp_internal_link_attr_name	
Description	The internal link reference attribute

Sample Markup	Possibly set through:
	`<Specials>`
	`<Link`
	`element="link"`
	`idref="linkend"/>`
	`</Specials>`

Notes	For non-DITA document types, you can identify the attribute with this ACL function: `link_idref_attr_name(`*`tagname`*`[,` *`doc`*`])`. This attribute is defined either in the `.dcf` file or in the document type. Special processing occurs for DITA documents to specify required attributes. This placeholder is used internally and dynamically in place of hard coded attribute names. This makes the MapObject more generic.

aticp_external_link_attr_name	
Description	The external link reference attribute

Sample Markup	Possibly set through:
	`<Specials>`
	`<Link`
	`element="ulink"`
	`uri="url"/>`
	`</Specials>`

Notes	For non-DITA document types, you can identify the element with this ACL function: `link_idref_attr_name(`*`tagname`*`[,` *`doc`*`])`. This attribute is defined either in the `.dcf` file or in the document type. Special processing occurs for DITA documents to specify required attributes. This placeholder is used internally and dynamically in place of hard coded attribute names. This makes the MapObject more generic.

As with the automatically generated MapTemplate files, you can modify a copy of the base files in the MapTemplate Editor as required for your site. Put the modified files in either the *`Arbortext-path`*`\custom\lib` directory or in a `custom\lib` directory referenced through the *APTCUSTOM* environment variable. Your customized MapTemplate file will override the base file.

Note

If you put a customized, automatically generated MapTemplate file (such as `rtf-topic.std`) into a `custom\lib` directory, then any changes you make to the base MapTemplate files that affect the RTF source type will not apply for that document type.

Following are some examples of how you might modify the base MapTemplate files:

- To expand the functionality of one of the base MapTemplate files, insert the appropriate placeholder name in place of actual document type dependent element names. Your base MapTemplate file will affect all document types. For example, if your Microsoft Word documents use special style names for division titles, you might add new MapObjects or modify the Input Rules of existing MapObjects. These MapObjects in a base MapTemplate file can include placeholder names for division elements and their title in the Output Steps of the MapObject's Output Rules.
- The base table MapTemplate files produce hard-coded element names rather than placeholder names, because these names are typical of the table models supported by Arbortext Editor. If your document type uses namespaced elements for one or more of the supported table models, you can modify the Output Rules of the base table MapTemplate and the change will affect all source types and document types at your site.
- Because default styles are less common in Adobe FrameMaker, custom MapTemplate files can provide better support for both divisions and their titles and numbered and bulleted lists.

Customizing the MapTemplate files

You can customize the MapTemplate files in the following ways:

- The base MapTemplate files themselves

  Changes you make here will affect all of the automatically generated files. Also, if Arbortext changes the base files in a future release those changes will not be reflected in your customized file.
- Custom overrides to the base files

  If there are just certain areas of the base map files that you want to customize, you can create a custom override file to the base files that just affects certain parts of the base file. Custom override files are named the same as the base file with `-custom` appended to the name of the file. For example, the override file for the `common-base.std` file would be named `common-base-`

`custom.std`. The changes you make in the custom override file are prepended to the base file and override the base file content. As with other customized files, you put the modified files in either the *`Arbortext-path`*`\custom\lib` directory or in a `custom\lib` directory referenced through the *APTCUSTOM* environment variable.

- The automatically generated MapTemplate files

  Changes you make here will override any customizations you make to the base files or a custom override file for the base files. In fact, if you have a customized file in place at this level, no automatic file generation takes place for that type of file.

  Again, if Arbortext changes the base files in a future release those changes will not be reflected in your customized file.

- Custom overrides to the automatically generated files

  As with the base files, you can create a custom override file for the automatically generated files that just affects parts of those files. This override file uses the same naming convention as the overrides to the base files. For example, the override file for `rtf-topic.std` would be named `rtf-topic-custom.std`. This override file would also need to be in a `custom` directory.

  Note that any changes you make in this override file would take precedence over changes you have made to the base files or a custom override file for the base files.

The best practice for customizing the MapTemplate files is to use the custom override files. This enables you to just change the specific parts of the base and automatically generated files that you need to modify. Note that this feature only recognizes the map objects found in a custom map template. It ignores the metadata in a map template, such as the drivers, driver options, and pre- and post-processing options. Those settings are defined in the `rtf-base.std`, `mif-base.std`, `htm-base.std`, and `txt-base.std` files.

The following sample files are available in the *`Arbortext-path`*`\samples\copypaste` directory:

- `common-base-custom.std` — Shows how to customize the default pasting behavior by having bold text in other applications converted to a processing instruction.
- `rtf-task-custom.std` — Shows how to use the **Paste Special** dialog box to paste numbered lists as steps and substeps in a DITA task.
- `pasteAsList.xlf` — Used with the `rtf-task-custom.std` sample file to show the changes to the **Paste Special** dialog box.

- `rtf-topic-custom.std` — Shows how to paste footnotes and index terms from Microsoft Word to a DITA topic.
- `rtf-axdocbook.std` and `pasteaslist.xlf` — Shows how to paste footnotes and index terms from Microsoft Word to an XML DocBook document.

A `readme.txt` file in the directory describes how to implement the sample files.

Manually generating the MapTemplate files

You can use the `create_copypaste_map` function to manually generate a MapTemplate file for a given source type and document type. The function returns the same MapTemplate file as the one automatically created by Arbortext Editor.

This function has the following syntax:

`create_copypaste_map` (*`source_type`*, *`path`*[, *`doc`*])

This function generates a basic MapTemplate file for the given clipboard source type and document type. It has the following parameters:

- *source_type* — Specifies the type of clipboard data. The following values are supported:
  - `htm` — HTML markup
  - `mif` — Maker Interchange Format (MIF)
  - `rtf` — Rich Text Format (RTF)
  - `txt` — Unicode and 8-bit ANSI text
- *path* — The full path to the generated MapTemplate file.

  The manually generated MapTemplate file must follow the same naming convention as the automatically generated files: *`source_type`*`-`*`doctype_name`*`.std`. For example, a MapTemplate file for the RTF source type and the axdocbook document type must be named `rtf-axdocbook.std`.
- *doc* — Optional. The identifier of the document for which the associated document type should be used for the MapTemplate file.

  If *doc* is omitted or is `0`, the current document is used.

The function returns `0` on success or one of the following error codes on failure:

Code	Values
1	Base MapTemplate file not found
2	Cannot open destination MapTemplate file
3	Invalid document type (such as ASCII)
4	Base table MapTemplate file not found

Code	Values
5	No DITA `DOCTYPE` element found
6	Unknown error creating MapTemplate file

Implementing Copy and Paste for a Custom Document Type

You can implement the copying and pasting from other applications feature for custom document types. However, to get the best results you must be sure to provide the configuration information in your document type that Arbortext Editor requires to be able to correctly convert the source clipboard data into your document type's markup.

Some of this information is in the document type itself, such as the supported table models. Also, some information is in the associated Arbortext Styler `.style` file. However, most of the required information is in the associated document type configuration (`.dcf`) file. You will get the best results from this feature by ensuring that this information is configured in your `.dcf` file. The following table summarizes the required `.dcf` file settings and provides example markup from the `axdocbook.dcf` file.

Description	Sample axdocbook .dcf File Markup

Primary paragraph element	`<Specials>`
	`<Paragraph`
	`element="para"/>`
	`</Specials>`

Highest division element	`<PasteOptions>`
	`<PasteElement`
	`category="primary_division"`
	`element="chapter"/>`
	`</PasteOptions>`

Primary division element	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect1"`
	`primary="yes"/>`
	`</ElementOptions>`

Nested primary division element(s)	`<ElementOptions>`
	`<ElementOption`
	`category="division"`
	`element="sect2"`
	`primary="yes"/>`

Description	Sample axdocbook .dcf File Markup
	`</ElementOptions>`

Primary division title element	`<PasteOptions>`
	`<PasteElement`
	`category="primary_division_title"`
	`element="title"/>`
	`</PasteOptions>`

Markup for bold text	`<TextStyles>`
	`<Bold`
	`attribute="role"`
	`attributeValue="bold"`
	`element="emphasis"/>`
	`</TextStyles>`

Markup for italic text	`<TextStyles>`
	`<Italic`
	`attribute="role"`
	`attributeValue="italic"`
	`element="emphasis"/>`
	`</TextStyles>`

Markup for underline text	`<TextStyles>`
	`<Underline`
	`attribute="role"`
	`attributeValue="underline"`
	`element="emphasis"/>`
	`</TextStyles>`

Markup for smallcaps text	`<TextStyles>`
	`<SmallCaps`
	`attribute="role"`
	`attributeValue="smallcaps"`
	`element="emphasis"/>`
	`</TextStyles>`

Markup for subscript text	`<TextStyles>`
	`<Subscript`
	`element="subscript"/>`
	`</TextStyles>`

Markup for superscript text	`<TextStyles>`
	`<Superscript`
	`element="superscript"/>`
	`</TextStyles>`

Markup for bulleted list	`<Lists>`
	`<Bulleted>`
	`<Block element="itemizedlist"/>`
	`<Item element="listitem"/>`

Description	Sample axdocbook .dcf File Markup
	</Bulleted> </Lists>
Markup for bulleted list item	<Lists> <Bulleted> ... <Item element="listitem"/> </Bulleted> </Lists>
Markup for numbered list	<Lists> <Numbered> <Block element="orderedlist"/> <Item element="listitem"/> </Numbered> </Lists>
Markup for numbered list item	<Lists> <Numbered> ... <Item element="listitem"/> </Numbered> </Lists>
Markup for internal document links	<Specials> <Link element="link" idref="linkend"/> </Specials>
The reference attribute for internal document links	<Specials> <Link element="link" idref="linkend"/> </Specials>
Markup for external links	<Specials> <Link element="ulink" uri="url"/> </Specials>
The reference attribute for external links	<Specials> <Link element="ulink" uri="url"/> </Specials>
Markup for block graphics	<Specials> <Graphic

Description	Sample axdocbook .dcf File Markup
	element="graphic" ... /> </Specials>
The height attribute for block graphics	<Specials> <Graphic ... reproDepth="depth" ... /> </Specials>
The width attribute for block graphics	<Specials> <Graphic ... reproWidth="width" ... /> </Specials>
The scalefit attribute for block graphics	<Specials> <Graphic ... scaleToFit="scalefit" ... /> </Specials>
Markup for inline graphics	<Specials> <Graphic element="inlinegraphic" ... /> </Specials>
The height attribute for inline graphics	Same attribute as block graphics
The width attribute for inline graphics	Same attribute as block graphics
The scalefit attribute for inline graphics	Same attribute as block graphics
Markup for the document type's default table model	<PasteOptions> <PasteElement category="primary_table_wrapper" element="informaltable"/> </PasteOptions>

Description	Sample axdocbook .dcf File Markup

The reference attribute for graphics	`<Specials>`
	`<Graphic`
	`...`
	`entity="entityref"`
	`filename="fileref"`
	`... />`
	`</Specials>`
	The determination of whether this is a file reference or entity reference depends on either the `.style` file or the `.dcf` file settings for graphic details. If the document type has a `.style` file, those settings take precedence and any `.dcf` file settings are ignored.

The ID attribute	`<Options`
	`...`
	`idAttribute="id"`
	`... >`
	`</Options>`

Arbortext Editor will use the information in your document type and the associated `.dcf` and `.style` files to build the automatically generated MapTemplate files for your document type and the relevant clipboard source formats. If you make a change to the `.dcf` file, then the automatically generated files are regenerated in the next copy and paste operation using the document type. If a `.style` file is associated with the document type, the information in the `.style` file for graphics, links, and link targets takes precedence over related `.dcf` information. All other information that controls copy and paste is collected from the `.dcf` file.

You might also want to customize the base MapTemplate files or the **Paste Special** dialog box for your custom document type. However, note that a MapTemplate file specific to a source type and document type (such as `rtf-task.std` for a DITA Task document) can be modified in the MapTemplate Editor without regard to the `.dcf` file information. Remember that the automatically generated MapTemplate files will be populated by placeholder element names rather than actual element names. You can modify the default MapTemplate files as desired using Arbortext Import, or even create custom MapTemplate files for copy and paste from scratch using the MapTemplate Editor. As long as you follow the file naming convention and put the customized MapTemplate file in a `custom\lib` folder, that file will control the conversion process.

Customizing the Paste Special Dialog Box

Besides the automatic copy and paste support, you can control how content from other applications is pasted into a document through the **Paste Special** dialog box. After you have copied content from another application, you can select **Edit ▸ Paste Special** to open the dialog box and select how you want the content to be pasted into your document.

By default, the **Paste Special** dialog box enables you to paste clipboard content in the following formats:

- **bullet list**
- **numbered list**
- **paragraphs**
- **table (from tabular text)**
- **table with no title**
- **table with title**
- **text**
- **title**
- **web link**

You can customize this dialog box to change the supported document formats, change the format to a different MapObject, or add support for elements in your custom document type. The supported document formats are configured in the following file: *Arbortext-path*`\lib\locale\en\pasteAsList.xlf`. Put your customized `pasteAsList.xlf` file in `custom\lib\locale\en` (or in your locale specific subdirectory).

The `pasteAsList.xlf` file uses the XML Localization Interchange File Format (XLIFF) document type. See www.oasis-open.org/committees/xliff/documents/xliff-specification.htm for more information about XLIFF. The XLIFF document type specifies sets of `source` and `target` tags, where the `source` tag contains the string to be translated and the `target` tag contains the string to be displayed in the dialog box.

Following is the markup from `pasteAsList.xlf` for the **bullet list** document format type:

```
<trans-unit id="1" resname="bullet_list">
<source>bullet list</source>
<target>bullet list</target>
</trans-unit>
```

Note the `resname="bullet_list"` part of this markup. That is a reference to one or more MapObjects in the `common-base.std` base MapTemplate file. These MapObjects are used to convert the content on the clipboard into the

document format specified by the selection in the **Paste Special** dialog box and take precedence over the ones used for the regular copying and pasting operations in that case.

The MapObject in `common-base.std` that uses the `bullet_list` reference must use an `EVAL_XPATH` Input Rule which is triggered by the following XPath expression:

```
//TEXTXMLSTREAM/HEAD/XYZDOCUMENTMETATAGS/XYZMETA[@name="STDSourceFileName"]
 [@value="bullet_list.aticp:source_type"]
```

This Input Rule can be expressed as "Match this MapObject if the source file name (an abstract name created from the *resname* attribute value) is equal to `bullet_list.rtf` or `bullet_list.mif` or `bullet_list.htm`" and so forth. Any MapObjects with Input Rules that match if the user has chosen the **bullet list** item in the **Paste Special** dialog box will perform the special conversions defined by those rules.

You can use Arbortext Editor to edit the `pasteAsList.xlf` file to add or remove document formats and to change the strings displayed in the **Paste Special** dialog box. You can use Arbortext Import to modify the MapObjects referenced in `common-base.std` (or any other MapTemplate) from `pasteAsList.xlf` or to add new MapObjects for new document formats.

Note that the MapObject priority as described in Arbortext Import online help is important to **Paste Special** customization. MapObjects in `common-base.std` have higher priority values, which means they have lower priority in the top-down MapObject selection process in the final generated map. A very specific MapObject such as those used for **Paste Special** might never be selected if that MapObject is located below more generic MapObjects that could match the selection first. Be sure to place **Paste Special** MapObjects near the top of your custom MapTemplate file.

Following are some examples of **Paste Special** customizations:

- Develop a **Paste Special** MapObject for a paragraph to convert a simple paragraph to a document type specific footnote element.

  To do this, follow these steps:

  1. Create a new `trans-unit` element in `pasteAsList.xlf`:

     ```
     <trans-unit id="n" resname="footnote">
     <source>Footnote</source>
     <target>Footnote</target>
     </trans-unit>
     ```

     Replace *n* with an appropriate integer number for the file.

  2. Create a new MapObject in `common-base.std` (or any customized MapTemplate file) with Input Rules that match the following XPath expression for a general purpose rule:

     ```
     //TEXTXMLSTREAM/HEAD/XYZDOCUMENTMETATAGS/XYZMETA[@name="STDSourceFileName"]
      [@value="footnote.aticp:source_type]"
     ```

Or that match the following expression for a rule specific to the RTF source type:

```
//TEXTXMLSTREAM/HEAD/XYZDOCUMENTMETATAGS/XYZMETA[@name="STDSourceFileName"]
 [@value="footnote.rtf]"
```

3. Add footnote-specific output steps to the newly-created MapObject to produce the footnote markup for your document type.

- Develop a new **Paste Special** MapObject to convert numbered lists into `steps` and `substeps` elements in the DITA Task document type, instead of the default behavior of converting numbered lists in the clipboard source into numbered lists in the document type markup.
- Develop a new **Paste Special** MapObject to convert Microsoft Word or Adobe FrameMaker index terms into document type specific index term markup.

Limitations

The copying and pasting from other applications feature has the following limitations:

- This feature only supports pasting into the Arbortext Editor Edit pane or Arbortext Styler Generated Text Editor.
- This feature does not affect copying and pasting markup between two Arbortext Editor windows.
- Because Arbortext Import/Export does not currently support text directionality, this feature does not support directionality.
- The quality of the pasted markup depends primarily on the structure and clarity of the source documents. In the case of Microsoft Word and Adobe Framemaker documents, using paragraph styles is important to successful and efficient conversion. If all paragraphs in the source document use the same style name, even though the formatting may create the illusion of division titles, it is not possible to automatically determine the desired markup. If the source document uses unique style names to represent division titles at various nested levels, then the paste results will be more automatic. Well-styled source documents can significantly reduce the time spent for manual cleanup.
- The quality of the pasted markup also depends on how well the document type elements have been defined in the document type configuration (`.dcf`) file, and to some extent the `.style` file.
- Support for inline styles (bold, italic, and so forth) requires an element that is unique to that purpose, such as `emphasis` with attributes, or `b` (for bold). Multiple, simultaneous inline styles, such as bold-italic, assumes the nested element model like `axdocbook`. By default, the feature can handle all

combinations of bold, italic, and underline. More sophisticated conversion requires a custom MapTemplate.

- While images can be copied and pasted from Microsoft Word, those images that have been constructed from a loose collection of drawing objects cannot be pasted directly as an image because it is not a single image.
- Adobe FrameMaker does not include image information in the clipboard data, so images cannot be copied and pasted from FrameMaker.
- If you select and copy a table from Adobe FrameMaker 7.2 or earlier, the table structure is not copied to the clipboard. However if you copy a sibling paragraph with the table (the paragraph before or the paragraph after), the table structure is converted correctly. This is a limitation of FrameMaker versions prior to 8.0.
- If you select and copy a table that contains another nested table and attempt to paste that table into a Arbortext XML DocBook document, a DITA document, or a document of another document type that requires that a table be enclosed in paragraph tags, the **Invalid Paste Structure** dialog box opens. Arbortext Editor does not enclose a nested table in paragraph tags in this case.

10

Customizing DITA Support

Customizing DITA support

Arbortext Editor provides a specialized user interface for editing DITA (Darwin Information Typing Architecture) maps and topics. You can customize parts of that interface to meet the needs of your site.

Customizing the DITA Resource Manager

The **Resource Manager** dialog box enables you to manage the references inserted into DITA documents. The **Resource Manager** stores information about its current state in the `arbortext.wcf` preferences file, so that state can be restored in future sessions. This information is stored in persistent user settings that can be customized to set a desired state at start up using an Arbortext Command Language (ACL) script or Arbortext Object Model (AOM) program. For ACL scripts, you set the preferences with the `set_user_property` function. For AOM programs, you use the `Application.getUserProperties` method.

Note that modifying a persistent user property has no effect on currently open **Resource Manager** dialog boxes. If a preference is modified while the **Resource Manager** is open, that preference might be overwritten by the dialog box when it closes. It is recommended that you set these preferences in a script or program located in the `custom\init` directory. That ensures the dialog box is not open at the time the preferences are set. You can also use the `dita_reset_rm_state` function before specifying your own settings to reset the **Resource Manager** state and ensure your settings are used. However, this function does remove all of the **Resource Manager**'s current state.

Customizing the displayed Resource Manager tabs

You can customize the tabs that appear in the docked version of the **Resource Manager** using the following persistent user preference:

`com.arbortext.dita.rm.(`*`doctype`*`).tabs=`
`(`*`tab1`*`),(`*`tab2`*`),(`*`tab3`*`),...`

For the *doctype* token, you can specify either a specific document type base name (for example, `ditabase` or `bookmap`), or you can use `map` to specify all DITA map specializations and `topic` to specify all topic specializations.

For the *tab* token, you can specify one of the following values:

***tab* Value**	**Tab Name**	**Valid Document Types**
`link_xref_tab`	Link/Xref	DITA topics
`image_tab`	Image	DITA topics and maps
`conref_tab`	Content Reference	DITA topics and maps
`topic_tab`	Topic	DITA maps

tab Value	Tab Name	Valid Document Types
`new_topic_tab`	New Topic	DITA maps
`keydef_tab`	Key Definition	DITA maps
`xinclude_tab`	Inclusion	DITA topics and maps

If no preferences are specified, the following default values are used:

```
com.arbortext.dita.rm.topic.tabs    link_xref_tab,image_tab,conref_tab

com.arbortext.dita.rm.map.tabs      topic_tab,new_topic_tab,keydef_tab
```

For example, you could use the following ACL code to remove the **Content Reference** tab from the docked **Resource Manager** dialog box for DITA topics and replace it with the **Inclusion** tab:

```
set_user_property(\
 'com.arbortext.dita.rm.topic.tabs', \
 'link_xref_tab,image_tab,xinclude_tab');
```

You could use the following ACL code to remove the **Key Definition** tab and add the **Image** tab for DITA BookMaps:

```
set_user_property(\
 'com.arbortext.dita.rm.bookmap.tabs', \
 'topic_tab,new_topic_tab,keydef_tab,image_tab');
```

Customing the Default Look in Location

You can customize the default location for the **Resource Manager Look in** option using preferences. Note that the setting of the **Synchronize Location Across Tabs** menu choice and associated `set ditasynctab` command also affects the **Look in** location. If tab synchronization is turned on, then any location you set at start up only applies until the user navigates to a different location in the **Resource Manager** browser.

You can use the following preferences to set the **Look in** location for specific tabs:

- `com.arbortext.dita.lastAccessDirForLinkXrefPane` — **Link/Xref** tab
- `com.arbortext.dita.lastAccessDirForImagePane` — **Image** tab
- `com.arbortext.dita.lastAccessDirForTopicContentReferencePane` — **Conref** tab on the **Resource Manager** for DITA topics

- `com.arbortext.dita.lastAccessDirForMapContentReferencePane` — **Insert Conref** dialog box for DITA maps.
- `com.arbortext.dita.lastAccessDirForKeyDefPane` — **Key Definition** tab
- `com.arbortext.dita.lastAccessDirForTopicsPane` — **Topic** tab
- `com.arbortext.dita.lastAccessDirForXincludePane` — **Inclusion** dialog box

For example, you could use the following ACL code to set the default **Look in** location for the **Link/Xref** tab to `C:\demo`:

```
set_user_property(\
 "com.arbortext.dita.lastAccessDirForLinkXrefPane", \
 "C:\\demo");
```

Customizing the Type of Files to Display in the Show and Type Options

You can customize the type of files displayed in the **Resource Manager Show** and **Type** options using the following persistent user preference:

`com.arbortext.dita.rm.[(`*`doctype`*`).](`*`tab`*`).(`*`filter`*`)`

For the optional *doctype* token, you can specify a specific document type base name (for example, `ditabase` or `bookmap`). If omitted, the preference applies for all document types that do not have an explicit preference value set.

For the *tab* token, you can specify the same values as those used to set the default tabs.

For the *filter* token, you specify the option you want to customize. You can use either `showFilter` or `typeFilter`.

The value of the preference is a string identifying the file filter. In general, the following naming rules are used for filters:

- When the filter is for elements based on a class, the value is that base class — for example, `topic/topic` for **All Topics**.
- When the filter is the same as a file extension or tag name, the name is the named thing in lower case — for example, `pdf`, `html`, `fig`, or `section`.
- Otherwise, the name is the same as the English label for the filter in lower case with spaces replaced by underscore (_) characters.

You can use the following values for **Show** option filters:

English Label	Value
All elements	`all`
All elements with IDs	`all_elements_with_ids`

English Label	Value
All topics	`topic/topic`
Key definitions	`key_definitions`
Key references	`key_references`
fig	`fig`
table	`table`
li	`li`
fn	`fn`
section	`section`
Valid Elements	`valid_elements`
Valid elements with IDs	`valid_elements_with_ids`

You can use the following values for **Type** option filters:

English Label	Value
Topic	`topic`
Topic or Map	`topic_or_map`
Map	`map`
Image	`image`
PDF	`pdf`
HTML	`html`
Any	`any`
All Graphics	`all_graphics`
Bitmap Graphics (*.bmp)	`bmp`
Graphics Interchange Format (*.gif)	`gif`
IsoDraw Graphics (*.iso, *.isoz)	`iso`
JPEG File Interchange Format (*.jpg)	`jpg`
Portable Network Graphics (*.png)	`png`
ProductView Graphics (*.edz, *.pvz)	`edz`
Scalable Vector Graphics (*.svg)	`svg`
Tag Image File Format (*.tif)	`tiff`
Vector Graphics (*.cgm, *.eps)	`cgm`

For example, you could use the following ACL code to set the default **Type** option value to **PDF** for all tabs and document types:

```
set_user_property(\
 "com.arbortext.dita.rm.typeFilter", \
 "pdf");
```

You could use the following ACL code to set the default **Type** option value to **Topic or Map** on the **Topic** tab for DITA BookMaps:

```
set_user_property(\
 "com.arbortext.dita.rm.bookmap.topic_tab.typeFilter", \
```

```
 "topic_or_map")
```

You could use the following ACL code to set the default **Show** option value to **All topics** on the **Link/Xref** tab for all Learning and Training topics:

```
set_user_property(\
 "com.arbortext.dita.rm.learningDitabase.link_xref_tab.showFilter", \
 "topic/topic")
```

Customizing the Tags Selected in the Insert Option

You can customize the tags selected in the **Resource Manager Insert** option using the following persistent user preference:

`com.arbortext.dita.rm.(`*doctype*`).[(`*tab*`).]preferredTags`

For the optional *doctype* token, you can specify a specific document type base name (for example, `ditabase` or `bookmap`). If omitted, the preference applies for all document types that do not have an explicit preference value set.

For the optional *tab* token, you can specify the same values as those used to set the default tabs. If omitted, the preference applies for all tabs.

The value of the preference is a list of tag names, separated by commas. As the cursor moves around the document, the **Resource Manager** scans this list until it finds a tag that is legal at the current location and make that the default selection in the **Insert** option.

For example, you could use the following ACL code to automatically select the `chapter` or `topicref` tags in the **Topic** tab **Insert** option for DITA BookMaps:

```
set_user_property(\
 "com.arbortext.dita.rm.bookmap.topic_tab.preferredTags", \
 "chapter,topicref")
```

The `part` tag still appears in the option, but is never selected automatically.

Customizing the Tags Displayed in the Insert Option

You can customize the tags displayed in the **Resource Manager Insert** option using the following persistent user preference:

`com.arbortext.dita.rm.(`*doctype*`).[(`*tab*`).]hiddenTags`

For the optional *doctype* token, you can specify a specific document type base name (for example, `ditabase` or `bookmap`). If omitted, the preference applies for all document types that do not have an explicit preference value set.

For the optional *tab* token, you can specify the same values as those used to set the default tabs. If omitted, the preference applies for all tabs.

The value of the preference is a list of tag names, separated by commas. As the cursor moves around the document, the **Resource Manager** scans this list until it finds the tags that are legal at the current location and displays only those tags in the **Insert** option.

For example, you could use the following ACL code to prevent displaying `topicset` or `topicsetref` tags in **Insert** option for DITA BookMaps:

```
set_user_property(\
 "com.arbortext.dita.rm.bookmap.hiddenTags", \
 "topicset,topicsetref")
```

Index

G

H

I

J

L

M

P

S

T

W