ptc

arbortext®

Arbortext Command Language Reference

8.1.2.0

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

Contents

1

Arbortext Command Language Overview

Arbortext Command Language (ACL) is flexible enough for you to perform simple operations and develop complex applications. ACL incorporates many C programming language concepts. A basic knowledge of the C programming language will be helpful, but not necessary. ACL is also similar to `awk` and `perl`, so knowledge of these programs can be useful.

The following are some examples of what you can do using ACL:

- write simple macros that increase user productivity
- map specific keys to work in specific windows or classes of windows
- redesign the menu system
- build and control your own panels and windows
- change the behavior of many “events” (for example, save, paste, modify tags, etc.) in the application
- communicate with other applications using the Arbortext Editor Application Programming Interface (API)
- build your own document management system

ACL Syntax Conventions

These are the conventions used to present ACL command syntax in this document:

- Command syntax statements appear in boldfaced type when they occur in the text flow.
- Square brackets ([]) indicate an optional item. The command is still complete if the optional item is left out. For example:

  ```
  quit [ok]
  ```

  The `quit` command opens a "save changes" prompt before you exit a document, while the `quit ok` command allows you to exit a document without opening a "save changes" prompt. (The square brackets are not part of the command.)
- A vertical bar (|) separates options when you can choose from several. Here is an example of an ACL command with more than one option:

  ```
  mark [begin | end]
  ```

  In the example above, the command can be `mark`, `mark begin`, or `mark end`. (The vertical bar is not part of the command.)
- Curly braces ({}) surround choices when they are not optional. You must supply one item from a list that is surrounded by curly braces. Here is an example:

  ```
  set expressions= {on | off}
  ```

  In the example above, the command is either `set expressions=on` or `set expressions=off`.
- Items that appear in italics are *tokens* that indicate the type of information you must supply at that location. Do not type these tokens verbatim. Here is an example of an ACL command that contains a token.

  ```
  insert_tag markupname
  ```

  In the example above, the command to insert a tag named `chapter` is `insert_tag chapter`.
- In path names, the term *`Arbortext-path`* refers to the directory in which Arbortext Editor is installed at your site. When you see *`Arbortext-path`*, replace it with your specific path name.

Note

You must separate multiple commands on the same line with a semicolon (for example, `copy_mark ii; save_buffers`). Also, vertical bars and curly braces that have a larger font size should be typed as part of the command.

This table summarizes the command syntax conventions:

Syntax conventions for commands

Command syntax conventions	
This font/notation:	**Indicates:**
boldface	a command syntax statement
italics	a token name for type of information required; do not type in verbatim (Example: insert_tag *markupname*)
[] (square brackets)	contents are optional
{} (curly braces)	contents are not optional; choose one if more than one choice is available
| (vertical bar)	separates command options
{} (larger curly braces)	include braces as part of your command
| (bold vertical bar)	include the vertical bar as part of your command

You can enter ACL commands at the Arbortext Editor command line. You can also place ACL commands in a separate file that is executed when Arbortext Editor is first started or when a particular document is opened.

Arbortext Editor Command File

The `_main.acl` file is the first command file Arbortext Editor reads during startup. This file is located in the `\packages` subdirectory of *`Arbortext-path`*.

Because `_main.acl` is read early in the Arbortext Editor initialization, before any documents are opened, commands that manipulate menus (for example, `menu_add` on page 674 and `menu_move` on page 679) are not valid in this file. Typically, the `_main.acl` file is maintained by a system administrator.

User Startup File

You can create a user startup file to store user-specific key mappings, aliases, or functions. Use the *APTRC* environment variable to specify your user startup file.

Following is an example of a user startup file.

Note

Arbortext Editor does not read lines that begin with the number symbol (#). They are comments.

```
#shows all markup
alias tf set tagdisplay=full;
#hides all markup
alias tn set tagdisplay=none;
#spell-checks selection
alias ck spe -selection;
#insert caption element
map alt+shift+c insert_tag caption;
#insert fig-block element
map alt+shift+f insert_tag figure-block;
#inserts template for corporate indexing
map f4 {insert_tag indexterm;insert_tag indextopic}
```

Document Type Command Files

Before Arbortext Editor opens a document, it loads any associated document type command files.

The *doctype*`.acl` file stores ACL code used to define the application. The *doctype*`.js` file stores JavaScript code to define the application.

Note

If you have both a *doctype*`.acl` file and a *doctype*`.js` file, Arbortext Editor will read *doctype*`.acl` before it reads *doctype*`.js`.

For example, in your *doctype*`.acl` file you might want to add a document type-specific directory to the `graphicspath` option using `append_graphics_path` in the *doctype*`.acl` file.

If you open another document of the same document type, Arbortext Editor will not reload the *doctype*`.acl` or *doctype*`.js` files.

Save the *doctype*`.acl` and *doctype*`.js` files in your `doctypes` directory.

Document Type Instance Command Files

Arbortext Editor reads document type instance command files each time you open a document incorporating the corresponding document type.

The `instance.acl` file stores document-specific ACL code. The `instance.js` file stores document-specific JavaScript code.

Note

If you have an `instance.acl` file and an `instance.js` file, Arbortext Editor will read `instance.acl` before it reads `instance.js`.

For example, if you add a new menu to the `instance.acl` file associated with the DocBook document type, every time you open a new or existing DocBook document, Arbortext Editor will read the `instance.acl` file and add the new menu to the document.

This file is located in the document type application directory.

Document Command Files

Arbortext Editor reads the document command files when you open a specific document.

The *`docname`*`.acl` file stores ACL code for a specific document. The *`docname`*`.js` file stores JavaScript code for a specific document.

Note

If you have a *`docname`*`.acl` file and a *`docname`*`.js` file, Arbortext Editor will read *`docname`*`.acl` before it reads *`docname`*`.js`.

Commands in the *`docname`*`.acl` and *`docname`*`.js` files override the commands in all previous command files.

You should save the *`docname`*`.acl` and *`docname`*`.js` files in the same directory as the *`docname`* document.

Note

If you save a document associated with one or more document command files under a new name, Arbortext Editor automatically creates new copies of those command files with the same name as the new document.

Displaying a Command History

At the Arbortext Editor command line, to see a history of commands previously typed, use **UP ARROW** or **DOWN ARROW**, or the default keymappings **CONTROL-P** or **CONTROL-N**.

Getting Online Help for Commands

Get online help for commands and command functions from the Arbortext Editor online help window. Also get help for commands or a specific command using the **Command** line.

Locating online help for a command or command function in the online help

Open the online help *Arbortext Command Language (ACL)* section. This section contains several section covering commands, functions, callbacks and so on. View the specific topics you're seeking help on.

Character Sets and Encoding

Arbortext Editor supports the import and export of both 8-bit (for example, ISO-Latin) and multibyte character encodings (for example, Shift-JIS and GB_2312-80). These character encodings are required for input, display, and publishing of the following languages:

- Japanese
- Simplified Chinese
- Traditional Chinese
- Korean
- English
- European (European characters are represented as character entities within Asian-encoded files)

Arbortext Editor supports the following character sets and encoding methods:

- Adobe-Standard-Encoding

  Adobe-Standard-Encoding allows a single font to be used for multiple operating systems.
- ISO-8859-1 through 9

  ISO 8859 defines accented and non-Latin characters used in European languages. ISO-8859 supports the following character sets:

  - Part 1: – Latin Alphabet No. 1

- Part 2: – Latin Alphabet No. 2
- Part 3: – Latin Alphabet No. 3
- Part 4: – Latin Alphabet No. 4
- Part 5: – Latin/Cyrillic Alphabet
- Part 6: – Latin/Arabic Alphabet
- Part 7: – Latin/Greek Alphabet
- Part 8: – Latin/Hebrew Alphabet
- Part 9: – Latin Alphabet No. 5
- Part 10: – Latin Alphabet No. 6

- ISO-10646-UCS-2

  The International Standards Organization's encoding that is equivalent to Unicode.

  ISO 10646 actually defines two encoding methods, one of which is a 4-byte form referred to as 10646.UCS-4. 10646.UCS-4 can encode over 4 billion unique characters.

  When writing an XML document in UCS-2, Arbortext Editor always writes out a byte order mark.

- EUC-JP

  Extended UNIX Code.

  EUC was developed as a method of handling multiple character sets, Japanese or otherwise, within a single text stream. EUC-JP is ISO-2022 compliant 8-bit encoding for which initially designated ASCII to GO and JIS X 0208-1983 to G1 without explicit announcement.

- Shift-JIS

  The Japan Industry Standard multibyte encoding.

  The encoding, developed by Microsoft, has been widely implemented on Japanese PCs. It is the Japan Industry Standard multibyte encoding. The codes are numerically shifted from the codes used by JIS standard X 0208. Shift-JIS is also referred to as MS Kanji and SJIS.

- Big5

  The multibyte encoding standardized by Taiwan. Traditional Chinese is encoded in Big 5.

- GB_2312-80

  The multibyte encoding standardized by the People's Republic of China. Simplified Chinese is encoded using GB_2312-80.

- KSC_5601

  The multibyte Wansung and Johab encoding standardized by Korea.
- UTF-8

  Unicode character set Transformation Format, 8-bit form.

  UTF-8 is a variable-length encoding of Unicode using 8–bit sequences. This transformation format was developed by X/Open. It is described in ISO/TEC 10646 AM1.
- US-ASCII

  American Standard Code for Information Interchange.

  ASCII specifies codes and SPACE and a set of 94 characters (letters, digits and punctuation or mathematical symbols) suitable for the interchange of English language documents.

2

Using the Arbortext Command Language

Testing and Debugging ACL Scripts

Use the `source` command to test ACL scripts. For example, if you have several ACL commands in a file called `pubcom.acl`, you can test the effect of these commands on Arbortext Editor by typing `source pubcom.acl` at the Arbortext Editor command line. If this file isn't in the current working directory, you will need to enter the path and file name. For example: `source test/pubcom.acl`

Note

If you are testing the contents of a command file, mapping the `source` command to a specific key makes it easy to repeatedly source that command file. See the `map` command for more info.

Running the `source` command actually runs all the commands in the `pubcom.acl` file as if you had typed them on the command line.

If the file that is sourced contains any syntax or other errors, a message appears explaining the nature of the error. This makes the `source` command useful for debugging ACL scripts.

Using Named Buffers

Ordinarily, each time you copy or cut text, you replace the contents of the default paste buffer with the text you just copied or cut. Named paste buffers provide a way to save and retrieve paste buffers that contain text passages that you expect to reuse multiple times.

Creating a Named Paste Buffer

To write a text passage to a named buffer, use one of these commands:

- `copy_mark` *buffername*
- `delete_mark` *buffername*
- `insert_string -buffer` *buffername*

Buffer names are case-sensitive within the Arbortext Command Language. Thus, you can create one buffer named bills and another named Bills.

Note

Unless the `-append` option is specified, any time you write to the named buffer, the current text replaces the contents of the buffer.

To insert the contents of a named paste buffer, use the command `paste` *buffername*. In this example, enter the following at the Arbortext Editor command line:

```
paste warntext
```

Creating a Named Current Buffer

If you want to create a named paste buffer to be the current paste buffer (the buffer that stores, and is replaced with each **Cut** and **Copy**), use the command `set paste=`*`buffername`* on page 861. The current paste buffer is also the buffer used by the `copy_mark`, `paste`, `delete_mark`, `read`, and `write` commands when no paste buffer name is specified.

Listing Paste Buffers

To list the named paste buffers in the current editing session, use the command `show buffers [output=`*`filename`*`]` on page 708.

To see a panel of the current named buffers, at the command line, enter:

```
sho buffers
```

or, to make a file of the names in the current directory, enter:

```
sho buffers output=buffers.txt
```

Inserting the Text of a Named Paste Buffer

The `paste` command allows you to insert text from a temporary or permanent (saved and loaded) named buffer at the cursor.

If `warntext` is a paste buffer you made in this editing session, to insert its text, enter the following command at the command line:

```
paste warntext
```

If `warntext` is a paste buffer you made in a previous editing session of this document, and you saved it with the `save_buffers` command to the document subdirectory, then to insert its text, enter this at the command line:

```
load_buffers
```

press **ENTER** and then enter this at the command line:

```
paste warntext
```

The text of `warntext` appears at the cursor.

Creating Buffers that Contain Files

Existing ASCII files or SGML files, such as Arbortext Editor documents or portions of them, can become the text of a paste buffer. To do this, use the `read -buffer` command. You can then use a `paste` command to put this file in your document.

To create a paste buffer named `cprt` that contains the text of an existing SGML file called `copyright`, enter the following command at the command line:

```
read -buffer cprt -SGML copyright
```

To paste the contents into the document, position the cursor where you want the text to appear and at the command line, enter:

```
paste cprt
```

To create a paste buffer named `pl` that contains the text of an ASCII file called `partslist`, enter the following at the command line:

```
read -buffer pl -untag partslist
```

To paste the contents into the document, position the cursor where you want the text to appear and at the command line, enter:

```
paste pl
```

ASCII files may require tagging when inserted into an SGML document.

Storing a Buffer from One Session to the Next

To save the contents of the named paste buffers from the current session for use in future editing sessions, use the `save_buffers` command or options on the `write` command: `-paste` or `-buffer` *buffername*. Otherwise, named paste buffers created during an editing session are removed when you exit Arbortext Editor.

The `save_buffers` command creates a *`buffername`*`.apb` file for each named buffer and puts these files in the specified directory. By default, the files are placed in the directory containing the document.

In this example, you created the named buffers `pl` and `warningtext` in this editing session and you want to save them for future use. At the command line, enter: `save_buffers`. This creates the files `pl.apb` and `warningtext.apb` in the user directory.

As another example, you want to save named paste buffers for a specific document type in a separate directory called `nambufs`. At the command line, enter:

```
save_buffers nambufs
```

Loading Saved Buffers

To load named paste buffers saved in a previous editing session, use the command. You may specify a directory from which to load.

Note

When you open a document, Arbortext Editor will not automatically load buffer files saved in the same directory as the document.

If the named paste buffers are in the directory with the document, enter the following command on the Arbortext Editor command line:

```
load_buffers
```

If the named paste buffers are in another directory, you must specify the path and file name.

Deleting Named Paste Buffers

To delete an unsaved named buffer during the current editing session, use the `delete_buffer` command. Otherwise, unsaved named buffers are deleted when you exit Arbortext Editor.

Note

The `delete_buffer` command does not remove files created with the `save_buffers` command. Use the `remove_file` command to delete files created with the `save_buffers` command.

Examples

To remove a named paste buffer, `partno`, which contains a particular equipment part number that you know you won't need again in this editing session, enter the following command on the Arbortext Editor command line:

```
delete_buffer partno
```

In this way, you are free to create another paste buffer named `partno` with a different equipment part number.

To delete all unsaved buffers, at the command line, enter:

```
delete_buffer -all
```

Example of Named Paste Buffers

Suppose you have a Arbortext Editor document that contains a list of unsorted addresses and you want to create two new separate sections—one for people in Maine and the other for people in Wisconsin. You could create two simple keymappings to help you with your task. The first keymapping, **F5**, would delete the highlighted address and add it to a paste buffer containing the addresses of people who live in Maine (ME). The second, **F6** would add them to a buffer for people who live in Wisconsin (WI):

```
map f5 {delete_mark -append ME}
map f6 {delete_mark -append WI}
```

After you go through the mailing list collecting the addresses for Maine and Wisconsin, paste in the sorted address lists by entering at the command line:

```
paste ME
```

and

```
paste WI
```

Modifying the Default Menus

With Arbortext Editor, it is possible to modify the default menus for an entire installation, for a particular document, for a particular user or group of users, or for a particular document. You may want to modify the order in which some items appear on the Arbortext Editor menus, delete menu commands, add custom functions to existing menus, or add new menus to the menu bar. It is also possible to replace the entire existing menu structure. You can also include shortcut menus for any window.

A single menu configuration file is used for all document types because there are no document-type specific items in the default menus. Arbortext Editor loads the file `editmenu.cf` from the `lib` subdirectory of *`Arbortext-path`*. A different default `editmenu.cf` file may be loaded by setting the environment variable *APTMENUFILE* to the path name of the menu configuration file.

The commands `menu_add`, `menu_change`, `menu_delete`, `menu_copy`, and `menu_move` let you customize the menus. The `menu_reset` command restores the menus to their default states. The `menu_save` and `menu_load` commands create and load a menu configuration file, which can be used to store your customized menus. (You need to maintain the correct path names to the menu configuration file if you move the document type to a new directory.)

For specific information on each command, click on the name of the command listed below:

`menu_add` on page 674
`menu_change` on page 676
`menu_load` on page 679
`menu_move` on page 679

menu_copy on page 678
menu_delete on page 678
menu_reset on page 680
menu_save on page 680

The type of menu customization determines which customization file it should be included in.

To modify the default menus for a particular document type, place the commands to modify or load menus in the document type instance command file on page 48.

For a document, place the commands to modify or load menus in the document command file on page 49 in the directory containing the document *docname* file.

To replace the default menu configuration file for all of the document types, set the environment variable *$APTMENUFILE* to the path name of the new menu configuration file.

To replace the default menu configuration file for a particular document type, use the menu_load command to load a menu configuration file, typically named menu.cf.

Caution

Arbortext does not recommend using menu_load regularly to configure your menu bar. The menu_load command causes the entire menu bar and all its items to be replaced. Therefore, Arbortext recommends adding, deleting, or changing only those items or menus necessary, while maintaining as much of the built-in functionality provided by the default menus as possible.

Arbortext Editor reads the document type command files (*doctype*.acl and *doctype*.js) and then the document type instance command files (instance.acl and instance.js) in the document type directory if they exist, before reading document command files (*docname*.acl and *docname*.js). The document type instance command files are read when the document type is loaded and whenever documents are switched. This allows document type-specific changes to the menus using the menu_add, menu_delete and menu_change commands, or another configuration file to be read using menu_load.

Because the document type instance command file is read each time you open a document of the same document type, any menu modification commands should be conditionalized using tests. For example:

```
if (!menu_exists(".MyMenu")) {
menu_add -menu . ".MyMenu"
menu_add ".MyMenu." "Some item" -cmd {some_command;}
}
```

Menu Paths

A menu path is a way of indicating the exact location of an item in the menu system. (This item can be either an item on a menu, or the menu itself.)

Menu paths look like directory paths, except periods are used to separate the components instead of slashes. The leading parts of a menu path correspond to a menu name, the trailing part matches a menu item. For example, `.File.New` refers to the item **New** on the **File** menu.

A menu path is considered absolute if it starts with a period (`.`). The name following the period must be the name of one of the top level menus on the menu bar. (For example, `.File`, `.Edit`, `.Tools`.) If a menu path does not start with a period, the entire menu bar hierarchy for the current window is searched for the first occurrence of the item starting with the first menu on the left and progressing down through every item on a menu before moving on to the next menu to the right.

You can use the menu path to specify a destination (as with the `menu_add` and `menu_move` commands). If a dot appears at the end of a menu path, it indicates the last item on the last menu named. For example, `menu_add .File.New. 'Create Invoice'` specifies that the new item **Create Invoice** would be added at the end of the **New** submenu of the **File** menu, but `menu_add .File.New 'Copy Document'` would cause the new item **Copy Document** to be added to the **File** menu after the item **New**.

When matching a menu path against a menu item, the mnemonic key character `&` is ignored in both strings. For example, "`.File.New`" will match the "`&File`" menu. Also, it is not necessary to specify a trailing ellipsis ("`...`") or mnemonic of the form "(`&X`)" on the target menu item. The menu path "`.File.Open`" will match the "`Open...`" menu item of the "`File`" menu. Similarly, "`.File.Exit`" will match the "`File`" menu item "`Exit(&Q)`". The trailing mnemonic (`&X`) is often used with Asian localized menus.

The syntax of a menu path allows specifications of a menu item by position and also by name. If a component of a menu path begins with `#` and is followed by one or more digits, then this specifies a numeric position. For example, the menu path `.File.#3` specifies the third item in the **File** menu. Numeric positions may be specified in any component. For example, `.File.#1.#2` is the same as `.File.New.Sample`, assuming the default menu configuration. The first non-title item is referenced by `#1`. The menu title, if any, may be specified by using `#0`. Blank or separator lines within the menu count as items. The function `menu_item_count` may be used to obtain the number of items in a menu.

Menu item labels may contain variable references. If a menu label contains any variable references, for example, `Modify $tagname`, the variable reference is substituted into the label string each time the menu containing the item is posted.

Shortcut Menus

You can create custom shortcut (popup) menus.

A shortcut menu is specified by using a colon as the first character of the menu path instead of a period.

To define a shortcut menu, specify : as the destination argument on the `menu_add` command. For example, this defines an abbreviated Edit menu, which could be posted in the Command window:

```
menu_add -menu : cmd -title "Command"
menu_add :cmd. Cut -cmd {delete_mark;} -key control-x
menu_add :cmd. Copy -cmd {copy_mark;} -key control-c
menu_add :cmd. Paste -cmd {paste;} -key control-g
menu_add :cmd. Delete -cmd {delete_mark null;}
```

Note

While the `-title` option is accepted, the title is not displayed at the top of shortcut menus.

The `menu_load` command may also be used to define shortcut menus. A shortcut menu is defined in the same way as a button menu, except the top-level definition for the shortcut menu is written using `PopupMenu` instead of `Menu`. A shortcut menu may not be referenced inside a `Buttons` section of the `menu.cf` file.

A user-defined shortcut menu is posted using the `menu_popup` function. The menu command posts the built-in menu for the current window. You need to map a key combination or mouse button to the function so users can access the menu. For example:

```
map m2 menu_popup("cmd")
```

where `cmd` is the name of a shortcut menu previously defined by a `menu_load` or `menu_add` command (as shown in the example above). The mapping could also be to a function that decides to post one of several shortcut menus based on the current position of the cursor or mouse.

A shortcut menu may be posted in any window. However, for performance reasons, it is often better to define a separate shortcut menu for each window. If a menu is posted in a different window than where it was last displayed, the window system objects in the previous window have to be destroyed and recreated in the new window.

The names of the built-in menus for the various window classes are shown in the table.

Built-in Menus For Window Classes

Menu name:	Class:
:EditPopup	edit
:Helpwin	help
:Workspace	helpwin[2-4], msgwin[1-4]

Special Characters in Menu Paths

A number of special characters can be used in the menu path to specify patterns, providing a compact way of expressing complex menu names.

As with regular expressions, most special characters can be preceded by a back slash (\) when you do not want them to be treated as special characters.

The special characters are listed below:

Special Characters In Menu Paths

.	Separates components of menu names. If it is the first character, it is an absolute path to a menu or item within the menu bar of the current window.
:	When the first character, refers to a shortcut menu.
*	Matches 0 or more characters. For example, file* matches file, filename, and file_pattern , among other things.
?	Matches any single character. For example, f?nd matches fend, find, fond, and fund, among other things.
`[...]`	Matches any one of the enclosed characters. For example, f[ie]nd matches find and fend, but not fond, fund, or fiend. You can use a hyphen within the brackets to indicate a range, such as 0-9, a-z, or A-z (for all letters, uppercase and lowercase).
_	Matches a space or an underscore when specifying a destination with the menu_add and menu_delete commands.

Using the Interface and Dialog Tools

The `message` and `readvar` commands and the `response` and `list_response` functions give the applications program a way to prompt for input from the user. You can also create non-blocking list selection dialog boxes using the `window_create` function.

The message Command

The `message` command is useful for displaying specific information to the user. Running the command displays it on the status bar or, if the message cannot fit on the status bar, it will be displayed in a dialog box.

The `message` command is asynchronous. This means that when a message command is issued in a script, it does not wait for the user to click **Dismiss** before executing the rest of the script. The script continues execution regardless of the action of the user on this message.

The `message` command does not return any value or signal and is therefore not useful for dialog purposes.

readvar and response

The `readvar` command and `response` functions can be used to prompt the user for input. `readvar` brings up a window with one or more entry fields in which text or values can be entered. The `-value` option enables the application programmer to present the user with a default value. If the user clicks on **OK** without entering a value, the default value is used.

The following two examples include a `readvar` with one entry field with a default value of `"Mr.X"` and a `readvar` with three entry fields.

```
readvar -prompt "Enter your name:" -value "Mr.X" name
readvar -prompt "Height:" -choice '5.5|8.5|11' h
-prompt "Width" -choice '8|11|16' w
-prompt "Depth" d
```

The first command prompts the user with only one entry field in which to enter his or her name. In the second example, there are entry fields with choice boxes for the height and width, and an entry box for the depth.

The `response` function creates a number of buttons in a special window, and returns a value indicating which button was clicked. The value of the button correlates to its place in the argument.

For example, the user is given three choices in the following example:

```
$choice=response("Are you ready?","yes","no","maybe")
```

When a user clicks on **yes** in the window, *$choice* is set to `1`. Clicking **no** sets *$choice* to `2`. Clicking **maybe** sets *$choice* to `3`.

list_response

The `list_response` function presents the user with a number of choices in a scrolling list panel. The list is passed as an array variable. For example, to create a list with four choices, enter on the Arbortext Editor command line:

```
split("red green blue black", $a)
```

```
$choice=list_response($a, "Choose a Color")
```

The sh Command

Information can be processed by either Arbortext Editor, using ACL, or it can be passed to the operating system shell for processing. Various features of Arbortext Editor allow you to pass information between the shell and Arbortext Editor.

The `sh` command passes the given command(s) to the system shell.

The following example displays the **Windows Character Map** window which allows you to insert extended characters not found on all keyboards:

```
sh charmap &
```

Use the ampersand (&) when launching any program that displays a window. The ampersand allows Arbortext Editor to continue processing while the window is displayed.

`sh` runs the commands in a new shell. This means that you cannot use the command to set environment variables in the login shell (`cmd.exe`).

When running a single command, quotes around it are allowed but not necessary.

```
sh ls -lag
sh 'ls -lag'
sh "ls -lag"
```

When running multiple commands, or commands that contain curly braces ({}), quotes are necessary and the commands must be delimited by semicolons (;) For example:

```
sh 'cd c:\temp;dir *.xml'
```

Note that single quotes inhibit variable substitution. When a variable is to be substituted, use double quotes. For example:

```
sh "cd %HOME%;del *.tmp"
```

Back-quotes

Information can be processed by either Arbortext Editor, using ACL, or it can be passed to the operating system shell for processing. Various features of Arbortext Editor allow you to pass information between the shell and Arbortext Editor.

The back-quotes (opening single quotes) perform a function similar to the `sh` command. Any command placed in back-quotes (` `) is also executed in a Bourne subshell. The difference is that the command returns what would normally be piped to standard output (as the `sh` command does). Therefore the result of the command has to be assigned to a variable. For example to evaluate the user name the following command can be applied:

```
$username =`whoami`message "$username"
```

As a second example, the following command extracts the current date from the system and assigns it to a variable:

```
$today=`date '+%y-%m-%d' `
```

This sequence runs the command `date '+%y-%m-%d'` in the UNIX shell and assigns the result to the variable *$today*. The value for this variable might be something like `"93-08-25"`.

Like the shell, runs of white space in the output (such as blanks, tabs, newlines, and so on) are replaced with a single blank character.

Using Regular Expressions

Regular expressions are used to construct simple to complex search criteria, as described in the following sections. Regular expressions are supported by the `find` on page 646, `substitute` on page 715, `caret` on page 621, and `window` on page 727 ACL commands as well as the `match` on page 411, `sub` on page 530 and `gsub` on page 378 ACL functions.

There are two ways to use regular expressions in commands. The first is to specify the `-e` option as part of your command (see the syntax for the individual command for more information). The second is to use the command `set expressions` on page 794`=on`.

Regular expressions are normally delimited by the `/` character when used in commands, although you may use any standard command string delimiters. When used as string terms in functions such as `match`, they must be delimited with the standard string term delimiters `"` or `'`. The rules for strings in double quotes apply before the regular expression is parsed: backslash sequences, such as `\n` (newline) and `\t` (tab), are replaced by the corresponding character and variable substitution is performed.

Special Characters

Most characters in regular expressions, such as alphanumeric characters, match the literal characters. For example, the `find -e /search/` command matches any occurrence containing “search”, such as “searching”, “searched”, and “research”. Special characters allow a regular expression to match more than one string. For example, a period (`.`) matches any single character. The command `find /s.t/ -e` finds “sat”, “set”, “sit” and any other three characters beginning with “s” and ending in “t”.

Other special characters include `^`, `$`, `?`, `+`, `*`, `|`, `[]`, and `( )`. These characters are discussed in the following sections.

Special characters can be preceded by a back slash (`\`) when you do not want them to be interpreted as special. For example, to interpret the period literally, place it after a `\`. For example, to search your document for numbers between 100 and 199 ending in “.5”, you would issue the command `find /1..\.5/ -e`. The

first two periods act as wildcards for any character, but the third period following the backslash is interpreted as a literal period. This command would also match strings with letters in them, such as `1ab.5`.

Character Classes

A character class is a regular expression constructed of characters enclosed in square brackets `[]` that matches any one character contained within it. For example, `[aeiou]` matches any one of the vowels listed between the brackets. Most special characters lose their special meanings within character classes. For example, `find /[.,]/ -e` would search for a literal period, a comma, or a space. To match a `]` character in a character class, place it as the first character inside the square brackets.

Some special characters used in other regular expression implementations are not supported: `\d`, `\D`, `\w`, `\W`, `\s`, and `\S`. Instead of these, you can use character classes such as `[0-9]`, `[^0-9]`, `[a-zA-Z0-9_]`, `[^a-zA-Z0-9_]`, and so on.

Negation

The circumflex character `^`, when used as the first character of a character class, searches for any matches other than the specification that follows it; that is, it matches any character not in the set. For example, the command `find /[^.,]/ -e` would match the next occurrence of any character other than a period, comma, or space.

Ranges

The hyphen (`-`) specifies a range when it appears as part of a character class. Use a hyphen to separate the first and last character of a range, such as `[a-z]` or `[0-9]`. For example, the command `substitute /[0-9]/#/ -e` replaces the next occurrence of any digit from 0 through 9 with the pound or hash sign (#).

To specify multiple digits, you would need to enter a numeric specification for each digit, such as `[0-9][0-9][0-9]` to find any three digit number. The command `find /1[0-9][0-9]\.5/ -e` would find numbers such as “123.5” and “154.5” but it would not match the string “1ab.5”.

To match a literal `-` as part of a character class, include it as the first or last character, for example, `find -e /[-0-9]/` would match a digit or negative sign.

Repetition

Search expressions can be expanded using the repetition characters `?`, `+` and `*`. These characters can appear after the specification to extend its match. A repetition character cannot appear as the first character of a regular expression.

- `?`

  Matches zero or one occurrence of the character or character class that precedes it. For example, `s[aeiou]?n` matched “sin”, “son”, “sun”, and “sn” but not “seen” or “soon”.

- `+`

  Matches one or more occurrences of a character or character class that precedes it. For example, `s[aeiou]+n` matches “sin”, “son”, “sun”, and also “seen” and “soon”.

- `*`

  Matches zero or more occurrences of a character or character class that precedes it. For example, `21*` can match “2”, “21”, and “2111”.

The repetition characters can also be applied to groups as explained in the following sections.

Alternation

The vertical bar `|` matches either the characters that precede it or the characters that follow it. For example, specifying `however|but` matches either word. You can group the alternative terms within parentheses and then refine the search as explained in the next section.

Grouping

You can use parentheses `( )` to group search criteria, such as `s(a|e)t`. The alternate specification `(a|e)` is a grouped term, producing matches for “sat” and “set”.

You can group an expression into a unit within a larger expression and then apply one of the repetition characters to the group, such as `?`, `+`, `*`, or the alternation character `|`. For example, `/java(script)?/` matches “java” as well as “javascript”. The regular expression `"(0|1)+"` matches a string of one or more `0` characters or one or more `1` characters.

For the replacement string to the `substitute` command or the `sub` or `gsub` functions, the `\n` combination can be specified to reference a specific group within parentheses in the matched result. The *n* represents the group number by counting groups from left to right. For example, `substitute '(John)(Smith)'\2,\1' -e` reverses the two groups as `Smith, John`.

The combination `\0` represents the entire matched result. For example:

```
substitute -e -nom 'tag'<\0>'
```

would surround the next occurrence of “tag” with angle brackets. The `-nom` option prevents the angle brackets in the replacement string from being interpreted as markup.

The `\n` combination can not be used in the regular expression itself.

Anchors and Word Boundaries

The following characters match a position in the string or paragraph instead of one or more characters:

- Placing the circumflex `^` before the search characters matches characters occurring at the beginning of a string or paragraph. For example, `/^Java/` matches strings that begin with “Java” but not ones that contain “Java” after the first character.

> **Note**
>
> If `^` appears first inside square brackets, it means negation.

- Placing the dollar sign `$` after the search characters matches characters occurring at the end of a string or paragraph. For example, `/y$/` matches anything that ends in “y”, and `/9$/` matches anything that ends in “9”.
- The combination `\b` matches a word boundary. For example, specifying `/\bJava\b/` finds the standalone word “Java” but not “JavaScript”.

> **Note**
>
> In the Arbortext implementation, the `[\b]` combination does not match a backspace character.

Because line breaks are not normally relevant or maintained when editing XML document content, the `^` and `$` anchor characters in a regular expression match paragraph instead of line boundaries when matching document content (for example, when using the `find` command). Paragraph elements are defined in the document type configuration file (`.dcf`) for the document type.

Searching for Tags or Entities

The `find` on page 646, `set` on page 744, `caret` on page 621, and `window` on page 727 commands include options that scan markup. To search for markup, you must enable markup scanning either by specifying the `-m` option for the command, or by entering `set markupscan` on page 908`=on` on the command line.

Arbortext Editor can recognize a start tag as <*element-name*>, an end tag as </*element-name*>, or an entity reference as &*entity-name*;. For example, the command `find -m '</para>'` would search for the next end paragraph tag, and `find -m '&prodname;'` would search for the next reference to the text entity named `prodname`.

Regular expressions can be used in the tag or entity name if you specify either the `-e` option as part of the command or if `set expressions=on` on page 794 option is in effect. For example, you could use the command `find -m -e '</.*>'` to search for the next occurrence of any end tag.

To search for missing punctuation before certain end tags you could use:

```
find -e -t '[^.?!]</(para|item)>'
```

Marking

Use the `mark` command on page 673 to mark a string in the document (this string may include tags) for further processing. Marking (or highlighting) is a way to select, evaluate, or remove certain parts of a document. The `mark begin` command initiates a selection, starting highlighting at the cursor. The `mark end` command terminates the highlighting and causes the region to be selected. This region ranges from the place where the cursor was (where `mark begin` was executed) to the place where the `mark end` is issued. At this point, the variable *$selection* is equal to the content of this region and can thus be evaluated.

In the following example, the cursor is first moved to the top of the document. Then it is placed after the first `chaphead` it encounters and the beginning of the selection is marked. Then the cursor is placed before the end tag (note the –1 value to place it before the end tag rather than after it). Then the end is marked. The content of the `chaphead` tag pair is then assigned to the variable *$title*.

```
caret first,first
caret 0,"<chaphead>" -t
mark begin
caret 0,"</chaphead>"-1 -t
mark end
$title = $selection
```

Manipulation of Selections

Three commands can manipulate a selected region.

- `copy_mark` — causes the selected region to be copied into the paste buffer.
- `clear_mark` — deselects a selected region.
- `delete_mark` — cuts the selected region and puts it into the paste buffer.

Usually, the `delete_mark` command is performed only when the region has balanced tag pairs. Unbalanced regions are allowed, however, if tags can be inserted such that the document remains in context.

Failure of any of these commands causes the *$status* variable to be set to `1`.

Command Variables

Arbortext Editor provides command variables in a manner similar to the UNIX shells. Command variables are used to specify a string whose exact value may not be known at the time. For example, suppose you want to make a command that tells you the current line number. You can't specify this value exactly because the current line number varies with the position of the cursor. To solve this problem, use a variable to indicate the cursor location.

Some variables are created within Arbortext Editor, while others are created outside Arbortext Editor as environment variables.

Variable Names

A variable is a “word” in a command that begins with a dollar sign ($) and a letter (such as $a), or a dollar sign and an underscore (such as $_). Variable names can be up to 100 characters long, not including the dollar sign ($). Examples of variable names are *$docname* and *$_currentline*.

Although a variable must start with a letter or underscore (_), the remainder of the name can consist of letters, digits, and underscores. Uppercase and lowercase letters are distinct; that is, *$ABC* and *$abc* are different variables.

Note

To specify a dollar sign ($) literally, not as part of a variable name, type `$$`.

Variable Assignment and Relation

By default variable does not need a special declaration. It is declared when you assign a value to it.

Note

You can use the `_STRICT_` predefined variable to force ACL to require that all variables in a package are explicitly declared. See Predefined variables on page 78 for details.

There are three types of variables in ACL: integer, string, and character. Each variable type is preceded by a dollar sign ($).

Below is an example of each variable type:

`$counter=5`	The variable is an integer.
`$name="Peter"`	The variable is a string.
`$char="c"`	The variable is a character.

Note

The initial value of every variable (that is, the value of an unassigned variable) is the null string ("").

ACL allows variable assignments that are similar to those used in the C programming language. For example:

`$i += 2`	is equivalent to	`$i = $i + 2`
`$i -= 2`	is equivalent to	`$i = $i - 2`
`$i *= 2`	is equivalent to	`$i = $i * 2`
`$i /= 2`	is equivalent to	`$i = $i / 2`
`$i %= 2`	is equivalent to	`$i = $i % 2`

Note that "/" is a division operator that uses integer arithmetic. This means that if `$a==5`, the result of `$a/3` is 1, not 1.6667.

The percent sign (%) is an operator that outputs the remainder of division arithmetic; thus, if `$a==5`, the result of `$a%3` is `2`.

ACL also allows abbreviated notation for increment operations. Here are some examples:

`$i = ++$n`	is equivalent to	`$n=$n+1; $i = $n;`
`$i = $n++`	is equivalent to	`$i = $n; $n = $n+1`

`$i = -- $n`	is equivalent to	`$n=$n-1; $i = $n`
`$i = $n --`	is equivalent to	`$i = $n; $n = $n-1`

Setting Values for Variables

An assignment statement of the format `$varname=expression` (where *expression* represents the new value for the variable) can be used to create a new variable and assign a value to it. You can also use this form of assignment statement to change the value of an existing variable. The expression must be terminated by the end of the command line or a semicolon (;). The assignment statement can be typed at the Arbortext Editor command line or entered in a command file (such as your *docname*`.acl` file). The new value will be recognized for the current editing session.

To change the value of an existing variable, you can:

- use an assignment statement to give the variable a new value
- reassign the value of the variable using an assignment operator (a subclass of expression operators)
- use the increment (++) or decrement (– –) operator

For example, you might use one of the following assignment statements to change the value of a variable:

```
$i++
$n += 5
```

The command `unsetvar` on page 725 and the `delete` on page 264 function allow you to “undo” the definition of a variable.

Note that the variables *$currentline*, *$dirselect*, *$selection*, *$tagname*, and *$topline* are read-only, and their values may not be changed.

Size Limits for ACL Variables

There is no size limit for a string-valued variable. However, the command parser limits the size of any command argument or string token to 4095 characters. For example, if both *$b* and *$c* are 2500 characters long, then the following example creates a variable that is 5000 characters:

```
$a = $b . $c
```

The variable *$a* shown above is too long to be substituted into a command at the Arbortext Editor command line. Therefore, typing the following would result in a parser error:

```
find "$a"
```

This variable could, however, be used in an expression or function call. Here is an example:

```
$d=substr($a,1500)
insert($a)
```

The `show variables` command gives you a listing of all defined variables and their values.

ACL Syntax Conventions for Quotation Marks

Either double (") or single (') quotes can be used to delimit strings.

When a variable occurs within double quotes, its value is substituted. For example,

String	Result

`$firstname="John"`
`$lastname="Smith"`
`$string1="My name is $firstname`
`$lastname."`
	My name is John Smith.

When a variable occurs in a string delimited by single quotes, it is treated as a literal string. For example,

String	Result
`$string2='My name is $firstname` `$lastname.'`	*My name is $firstname $lastname.*

If a dollar sign ($) is directly followed by characters, the sequence is treated as a variable string. To treat the sequence as a literal string, use two consecutive dollar signs. For example,

String	Result

`$firstname="John"`
`$lastname="Smith"`
`$string3="My name is $$firstname`
`$lastname."`
	My name is $firstname Smith.

Also note that backslash (\) sequences are recognized in double-quoted strings but not in single-quoted strings. In the following example, the backslash (\) and double quotes (“) are used to create a newline character that is assigned to the variable *$n1*.

```
$n1="\n"
```

The next example is a string of the two characters `\' and `n':

```
$n1=`\n'
```

String Concatenation

Use a dot (.) as the concatenation operator to merge two strings into one string. You can also use a dot to concatenate a combination of string variables.

The following example shows how dot concatenation is used to set the value of *$filename* to `datafile.txt`.

```
$textfile="datafile"
$extension=".txt"
$filename=$textfile . $extension
```

Note

The dot in `$extension=".txt"` is not a concatenation operator.

The following example shows how to use the dot operator to concatenate a combination of strings and string variables:

```
$firstname="John"
$lastname="Smith"
$myname="My name is " . $firstname ." " . $lastname . "."
```

In the example above, the value of *$myname* is *My name is John Smith*.

Symbolic Parameters

A symbolic parameter is a variable, such as *$1* or *$2*, whose value is defined by the context within which it is used. Symbolic parameters can be used to define arguments for aliases. For example, a generic alias can be used to replace a specific string in a document by a different string. If the alias is used more than once for different replacement strings, the following alias using symbolic parameters would be a solution.

```
alias replace_string {
 find $1
 delete_mark
 insert_string $2
}
```

When the command `replace_string "Peter" "John"` is run, the alias uses the first argument (`"Peter"`) in the `find` command and the second argument (`"John"`) in the `insert_string` command. The result would be that the first occurrence of the string `Peter` after the cursor (in the document) would be replaced by the string `John`.

Arbortext Editor allows you to supply parameters to the `alias` command. This means that you can define an alias, leaving a slot for information you will supply when you type the new command. This “slot” is indicated by the notation *$1*, *$2*,

$3, and so on, where *$1* is the location where the first piece of information encountered is to be supplied, *$2* is the location for the second piece of information, and so forth.

For example, if you have a tendency to type `open` at the command line as the command to open a different document, you can define the alias:

```
alias open edit $1
```

To move to the document `ProposalA`, you can type either `edit ProposalA` or `open ProposalA` at the command line.

$* Symbolic Parameter

*$** means substitute all the parameters supplied to the alias and *$0* refers to the alias name. For example, the command:

```
alias screendump print editor screen $*
```

defines a command to do a screen dump (print the screen contents) and (because of the *$**) allows users to specify print options with the `screendump` command. For example, you could print the screen contents by typing:

```
screendump
```

The following command would print the screen contents, but it would add page numbers to the header and a date stamp to the footer:

```
screendump header=pageno footer=datemark
```

You can also use symbolic parameters to substitute part of an option. For example, to define a command that opens (for editing) a document in the directory `/user/xxx`, issue the command:

```
alias homedir edit /user/xxx/$1
```

If you were editing a document in a different directory and issued the command `homedir todo`, your document would be replaced with the document `todo` from directory `/user/xxx`.

To simplify things even more, you can shorten the name of your new command from `homedir` to `hd` by typing:

```
alias hd homedir $*
```

To edit `todo`, enter `hd todo`.

The *$0* symbolic parameter will substitute the alias name within the alias invocation itself. For example:

```
alias testecho response("You just ran the $0 function.")
```

Running the `testecho` alias would generate the following response:

```
You just ran the testecho function.
```

$# Symbolic Parameter

The *$#* substitutes the number of arguments specified on the alias invocation. For example, if you entered the following alias:

```
alias argc {eval $#, "arguments";}
argc a b c
```

and executed the following command:

```
argc a b c
```

you would generate the following output:

```
3 arguments
```

$"*n* Symbolic Parameter

The *$" n* parameter refers to the *n*th positional parameter in the alias invocation, quoted as necessary to form a valid string term for expressions. This is most useful for providing an alias wrapper for a user-defined or built-in function. For example,

```
function xyz(s) { eval "xyz:", s output=>*;}
alias xyz { xyz($"1);}
```

allows the alias to be invoked as any of the following:

```
xyz test
xyz "test two"
xyz 'test 3'
```

Since aliases use the older convention of doubling string delimiters instead of the C-style backslash escape (that is, `\"`), you must double the delimiter to get an embedded quote. For example,

```
xyz "embedded ""quote"
```

would expand to

```
xyz("embedded \"quote")
```

$"* Symbolic Parameter

The *$"** symbolic parameter indicates that all arguments to the alias should be concatenated and then quoted as a single string term to replace the parameter. The quoting follows the same rules as for *$"n*.

Predefined Variables

Some variables are predefined by Arbortext Editor. This following is a list of predefined variables and what they represent. All of these variables are automatically maintained by Arbortext Editor.

- *$ENV* — An associative array in the main package defining the current environment. For example, to refer to the environment variable *APTCATPATH* in any package, use `main::ENV["APTCATPATH"]`. In the main package, *$APTCATPATH* can be used directly, although this use is not encouraged since it is not clear whether you are referring to a Arbortext Editor or system environment variable.

Changes to the *$ENV* associative array will be inherited by child processes. For example, the following expression changes the value of the *TMPDIR* environment variable.

```
$ENV["TMPDIR"] = "c:\arbortemp"
```

The environment is affected only if the associative array is changed by direct assignment. The environment will not be changed if the array value is modified by function (for example, used as a value in a call to the `chop` on page 242 function).

You can delete an environment variable by setting its value in the *$ENV* associative array to a null string. For example, the following expression removes the *TMPDIR* environment variable from the current environment without removing the *$ENV["TMPDIR"]* item in the array.

```
$ENV["TMPDIR"] = ""
```

To delete *$ENV["TMPDIR"]* from the *$ENV* associative array, use the following function:

```
delete($ENV["TMPDIR"])
```

- *$ERROR* — The syntax or run-time error message for the last command executed. It is set to the null string before any command or expression (including those from the `execute` on page 351 and `eval` on page 349 built-in functions) is compiled.
- *$ERRORS* — The array that holds the last few errors that were stored in *$ERROR*. The number of stored errors is determined by the value of *$ERRORS_SIZE*. An entry can added to this array either by an error in a function or command (such as `doc_open` on page 325) or by any ACL command or function which explicitly assigns a value to *$ERROR*.

  To clear all previous entries from *$ERRORS*, follow the `delete` on page 264 function example:

  ```
  delete(main::ERRORS)
  ```

  The *$ERRORS* variable is automatically recreated when the next error is generated.

  An example of a function to retrieve the array stored in *$ERRORS*:

  ```
  function get_last_errors()
  {
   local errors = "";
   if (defined(main::ERRORS)) {
   local lbound = low_bound(main::ERRORS);
   local hbound = high_bound(main::ERRORS);
   if (hbound >= lbound) {
   # Build a string with the errors (newest to oldest).
   local i;
   for (i = hbound; i >= lbound; i--) {
   errors .= main::ERRORS[i] . "\n";
  ```

```
}
}
}
 return errors;
}
```

- *$ERRORS_SIZE* — Controls how many errors are stored in *$ERRORS*. If a new entry to *$ERRORS* exceeds the limit of *$ERRORS_SIZE*, the oldest entry is removed. The default is 10.
- *$is_compact_install* — Determines if the version of Arbortext Editor running is the compact installation. If Arbortext Editor is the compact installation, returns any value other than zero. If Arbortext Editor is the full installation, returns zero.
- *$is_e3* — Determines if the Arbortext Publishing Engine is running in server-mode (no user interface) and, if true, returns any value other than zero.
- *$is_e3_interactive* — Determines if the Arbortext Publishing Engine is running in Arbortext Publishing Engine Interactive mode (has user interface) and, if true, returns any value other than zero. Used with *$is_e3*, as in the following example, to determine the mode in which the Arbortext Publishing Engine is running:

```
if (main::is_e3) {
 # I'm running in PE lights-out (server) mode.
}
else if (main::is_e3_interactive) {
 # I'm running in PE Interactive.
}
else {
 # I'm running in Editor.
}
```

- *$OFS* — The output field separator used by `eval` command to separate expressions. The default value is a single blank.
- *$ORS* — The output record separator used by `eval` command to terminate an expression list. The default value is a single newline.
- *$PCS* — The separator that separates parts of a path. It is a read-only variable equal to "\".
- *$PLS* — The path list separator, that is, the character that separates path names in a list of path names. It is a read-only variable equal to "`;`".
- *$RESPONSES* — The array that holds the last few responses that were stored in *$RESPONSE*. The number of stored responses is determined by the value of *$RESPONSES_SIZE*. An entry can be added to this array either by a response from the `response` on page 506 function, the `message` on page 681 command or by any ACL command or function which explicitly assigns a value to *$RESPONSE*.

To clear all previous entries from *$RESPONSES*, follow the `delete` on page 264 function example:

```
delete(main::RESPONSES)
```

The *$RESPONSES* variable is automatically recreated when the next error is generated.

An example of a function to retrieve the array stored in *$RESPONSES*:

```
Function get_last_responses()
{
 local responses = "";
 if (defined(main::RESPONSES)) {
 local lbound = low_bound(main::RESPONSES);
 local hbound = high_bound(main::RESPONSES);
 if (hbound >= lbound) {
 # Build a string with the responses (newest to oldest).
 local i;
 for (i = hbound; i >= lbound; i--) {
 responses .= main::RESPONSES[i] . "\n";
}
 return responses;
}
```

- *$RESPONSES_SIZE* — Controls how many responses are stored in *$RESPONSES*. If a new entry to *$RESPONSES* exceeds the limit of *$RESPONSES_SIZE*, the oldest entry is removed. The default is `0` for Arbortext Editor and `10` for the Arbortext Publishing Engine.
- *$OFS* — The output field separator used by `eval` command to separate expressions. The default value is a single blank.
- *$SYMTAB* — An associative array giving the symbol table for global variables in the current package. *SYMTAB[expr]* is equivalent to the variable named by the result of *`expr`*, for example, *SYMTAB["x"]* is a synonym for the variable `x`. Note, one way to test for the existence of a variable "var" is to use the expression `(var in SYMTAB)`. This is equivalent to `defined(var)`. *SYMTAB* is the only predefined variable which exists in all packages. All other predefined variables are only in the main package.
- *$appdata* — The path name of the application data directory. This is the same value as returned by the `get_appdata_dir` function called with no arguments. *$appdata* is a read-only variable.
- *$aptpath* — A read-only variable which specifies the location of the directory that contains the program files needed to run the software.
- *$arch* — A read-only variable that specifies the architecture of the host running the software.

- *$comproot* — The installation directory used for publishing. If Arbortext Publishing Engine is being used for publishing, this is the path name on the Arbortext Publishing Engine server. Otherwise, the value is the same as the value of `$aptpath`. *$comproot* is a read-only variable.
- *$currentcolumn* — The number of the column where the cursor is positioned. This variable is valid only if the current document is displayed in a window. This is a read-only variable.
- *$currentline* — The number of the line where the cursor is positioned. This variable is valid only if the current document is displayed in a window. This is a read-only variable.
- *$dirname* — The directory name of the current document if it is attached to an edit class window, or the document attached to the Edit window that last had focus.
- *$dirselect* — The name of the currently selected file name in the directory display (if no file name is selected, its value is null). This is a read-only variable.
- *$docname* — The name of the current document if it is attached to an `edit` class window, or the document attached to the Edit window that last had focus. It is the name without any leading path name components. For example: `/jdoe/mydoc.sgm` becomes `mydoc.sgm`.
- *$filename* — The path name of the file being edited including any leading path name components and the file name. It refers to the current document if it is attached to an `edit` class window, or the document attached to the Edit window that last had focus.
- *$home* — A read-only variable that holds the value of the user's home directory. *$home* defaults to the value of the *HOME* environment variable. Otherwise, *$home* uses the home directory as specified by the operating system. If no home directory can be determined, *$home* is set to the Arbortext Editor startup directory.

  The tilde character (~) can be used with ACL commands to represent the home directory. For example:
  ```
  edit -newwindow ~/report.xml
  ```
- *$progname* — The full name of the program being executed, for example, Arbortext Editor.
- *$selection* — A string representing the selected region in the current window (if no selection exists in that window, its value is null). If `sgmlselection=on`, the string includes any SGML codes. Otherwise, tags and other SGML are not included in the string. This is a read-only variable.
- *$sh_status* — The exit status of the process executed by the last `sh` command or backquote expression.

- *$status* — A number representing the status of the last command executed: `0` means successful completion. `1` means failure. `2` (which only applies to conditional commands) indicates a syntax error.
- *$system* — A string specifying the type of system on which Arbortext Editor is running. *$system* is a read-only variable.
- *$tagname* — The name of the tag to the left of the cursor in the Edit view (in ASCII documents its value is null). End tags start with slash (/). Angle brackets are not included in the value. This is a read-only variable.

> **Note**
>
> In the Document Map view, the process for determining *$tagname* is controlled by the `set docmapcurrenttag` on page 782 command.

- *$topline* — The vertical position in the document of the line at the top of the window. This variable is valid only if the current document is displayed in a window. This is a read-only variable.
- *$version_build* — The current build number of Arbortext Editor. This is a read-only variable.
- *$version_date* — The date of the current version of Arbortext Editor. This is a read-only variable.
- *$version_date_code* — The current software date code. For example, “M020”. This is a read-only variable.
- *$version_number* — The current version number of the software. This is a read-only variable.
- *$version_release* — The current software release number. For example, “5.4”. This is a read-only variable.
- *$winsys* — A string specifying the type of windows system for which this version of Arbortext Editor was created. (For example, "`Windows`".) *$winsys* is a read-only variable.
- *$wintype* — A string specifying the internal windows system implementation type for which this version of Arbortext Editor was created. (For example, "`MFC`" or "`Galaxy`".) *$wintype* is a read-only variable.
- *_STRICT_* — A variable that controls how variables are declared in ACL packages. If you declare the *_STRICT_* variable at the top of your package file, all variables must be explicitly declared. The default behavior is for variables to automatically declare themselves the first time they are referenced.

  If the statement `global _STRICT_` appears after the `package` statement in an ACL file, then the ACL interpreter will flag as errors or warnings any of the following variable uses.

- A reference or an assignment to a variable not previously declared on a `local` on page 667 or `global` on page 654 statement, or not declared as a formal argument to a function.
- An attempt to redefine a global variable with another `global` statement.
- A reference to a undeclared variable on a `readvar` on page 699 command.

An error is flagged only if you have `set stricterrors=on`. If the `stricterrors` option is set to `off`, a warning is displayed only if you have `debug` set to `1`. Otherwise, no error or warning is given.

Most implicit declaration warnings will be detected when the package is sourced. However, some will not be detected until the offending ACL statement is executed (for example, the variable specified on a `readvar` command).

A package file declaring the *_STRICT_* variable may be sourced more than once in a session. A global variable declared the first time a package is sourced will not be flagged as a duplicate definition the second and subsequent times the package is read. Once the "strict" behavior is turned on for a package, it cannot be turned off in the same session by undefining the *_STRICT_* variable or deleting its definition in subsequent versions of the package file.

The *_STRICT_* variable may not be declared in the main package as it would break existing ACL code. When developing new packages, use the *_STRICT_* directive as it's helpful for detecting mistakes in variable names.

Arbortext Editor also recognizes environment variables. The associative array *$ENV* is the preferred way to access environment variables. However, for variable references used in the main package, if Arbortext Editor cannot find a variable of the name you supply, and an operating system environment variable of the same name exists, its value is used.

Use the command `show variables` on page 713 or `show vars` to display a list of the variables defined in the main package.

Creating your Own Variables

You can use the `readvar` command, or an assignment statement (of the format *$ varname=expression*) to define your own variables. In addition, many functions will set variables.

Note

Arbortext Editor will prompt for a value any time a `readvar` command is encountered. This can be useful when you are writing a script and you want to prompt dynamically for the value of a particular variable.

Example: Mappings Using Variables

Some example mappings using variables are:

```
map control-s {execute cp $filename $filename.bak; \
save;}
```

This changes the **CONTROL+S** sequence so that it first makes a backup copy of the previous version of the file with a `.bak` extension and then saves the current version.

```
map f2 {readvar -prompt \
"Enter story name:" story; exec edit $story;}
```

This is much like the edit command without arguments, but allows a specialized prompt string.

Using Variables

To use a variable, supply it in the command at the point at which you wish its value to be used. For example, suppose you want to know which line of a document the cursor is positioned in. You can't supply the line number explicitly without counting all the lines in the file, but you can use a command containing the variable *$currentline* to determine this value:

```
message "line: $currentline"
```

Entering this command at the command line displays the line number on the status bar.

The execute Command

For most commands, variables are substituted when Arbortext Editor compiles commands into an internal representation rather than when the command is executed. (The exceptions are the `execute` and `eval` commands, the `if/else` and `for/while` conditional commands, function calls, and variable assignment where variables are substituted when the command is executed.) This can pose problems with the `map` on page 669, `alias` on page 617, and `print` on page 691 commands unless you used the `execute` command on page 645 to defer evaluation of any variables until the action is triggered.

Note

Variables will be substituted within strings that are quoted using double quotes ("), but not within strings delimited by other characters.

Arrays and Variables

An array is a special type of variable because it can contain more than one value. An array consists of an array name and an index. The index identifies the element of the array that is addressed.

Arbortext Editor provides two types of one-dimensional arrays:

1. normal arrays — are indexed by integer subscripts
2. associative arrays —are indexed by arbitrary strings

You do not have to declare arrays or array elements or how many elements are in an array. When used in expressions, arrays are created when referenced in a manner similar to scalar variables. For example, as long as you have not already defined the scalar variable *$a*, then `$a[1] = 'one'` creates the array *$a* and assigns the string "`one`" to element `1`.

Associative arrays are much like normal arrays except that subscripts are actually strings called keys. For example, `$aa["one"] = 1` assigns the value `1` to the element with the key "`one`". A null subscript of an associative array is equivalent to a key of "`0`".

You can iterate over all the keys in an associative array using the `for` statement:

```
for (key in array)
```

For example, the following commands iterate over the two elements of the associative array price:

```
$price["candy"] = 45
$price["pop"] = $price["candy"] + 5
for ($junk in $price) {
 $price[$junk] += 10;
}
```

A normal array is implemented internally using an array which dynamically expands if necessary as elements are added to it. An associative array is implemented using a hash table so lookups are fast regardless of the array size, but not as fast as a normal array. An associative array also usually requires more memory since each element has a fixed overhead due to the hash table.

Since the syntax for both normal and associative arrays is the same, Arbortext Editor decides which kind of array to use based on the subscript. If the subscript is an integer, a normal array is assumed. An associative array is used if the subscript

is a string. If the array subscripts are all small integers but then a string subscript is used, Arbortext Editor will convert the array into an associative array. For example, with

```
arr[1] = 1
arr[2] = 2
arr["three"] = 3
```

the array *arr* starts off as a normal array but when the third assignment is executed, the array gets converted into an associative array. Arbortext Editor will also convert a normal array into an associative array if the array becomes sparse, that is, the majority of the elements are undefined. For example, with

```
arr[1] = 1
arr[2] = 2
arr[100000] = 100000
```

after the third statement, the array would become an associative array since considerably less memory would be needed to represent it. If you use integer subscripts with an associative array, it is more efficient to use a string subscript as the first reference. For example, preceding both examples above with

```
arr[" "] = " "
```

would avoid the conversion into an associative array.

When used in an expression, the subscript of an array may be an expression. The resulting value is used to access the array element. However, when used in variable substitution, the subscript may only be an integer constant (or string if an associative array), or a scalar variable. For example, the following command will not parse:

```
caret $pos[$i+1]
```

because `$i+1` is an expression. This command can be rewritten, however, as

```
$i++;
caret $pos[$i]
```

Although you don't have to declare a global array variable, you can use the `global` statement to define it as follows:

```
global arr[]
```

The array *arr* can be either type of array, its subscripts determine whether it is a normal or associative array. The `global` statement can also be used to declare a fixed size normal array. For example:

```
global arr[10]
```

declares an array with 10 elements, indexed from 1 to 10. In this case, if you attempt to use a subscript not in this range, Arbortext Editor will issue an error message. You can specify a different starting subscript by using the notation `global arr [lb..hb]`. For example:

```
global arr[-5..5]
```

defines an array with the lower bound `-5` and higher bound `5`. If you know the size of an array, it is more efficient to declare it as a fixed array of that size instead of letting the array dynamically expand. Most of the built-in functions that return arrays return fixed-size arrays.

The `local` statement can be used to declare local array variables and has the same syntax as `global`. The `low_bound` and `high_bound` built-in function return the bounds of a fixed-size array. The `count` built-in function returns the number of elements actually in use in either a normal or associative array.

The `unsetvar` command can be used to undefine an array, discarding its elements. For example:

```
unsetvar arr
```

deletes array *$arr*. Note that no leading dollar sign is used on the `unsetvar` command.

To delete an array element, use the `delete` built-in function. For example:

```
delete($arr[$i])
```

Deleting an element releases the memory for it and marks it undefined, both for normal and associative arrays. The built-in function `count` will return one less after the deletion. For a dynamic (non-fixed size) normal array, if you delete the first element of the array, the `low_bound` function will return the next valid index. If you delete the last element, `high_bound` will return the previous index.

The `delete` function may also be used to undefine an array, as in:

```
delete($arr)
```

Using Expressions

Expressions are used by the `if/else`, `for/while`, `switch`, and `eval` commands. Expressions are also used in function calls and in assigning variables.

For commands with expressions, evaluation of variables within the expression is deferred until the command is executed. With other commands, variables are substituted when the command is compiled, unless the `execute` command is used to defer evaluation.

Expression Operator Precedence

Expressions are built using the following operators, which are listed in increasing order of precedence (grouping operators have highest precedence; assignment operators have lowest). Within each table, the operators have equal precedence.

- Assignment operators on page 89
- Conditional expression operator on page 90
- Conditional Logical “or” operator on page 90

Assignment

It is important to note that the order in which items appear in this table are arbitrary. All operators have equal precedence to ACL.

Expression operators assignment table

`=`	`$i = 1` sets the value of *$i* to 1.
`+=`	`$i += 2` increases the value of *$i* by 2. (Same as `$i = $i + 2`)
`-=`	`$i -= 2` decreases the value of *$i* by 2. (Same as `$i = $i - 2`)
`*=`	`$i *= $n` increases the value of *$i* by a factor of the value of *$n*. (Same as `$i = $i * $n`)
`/=`	`$i /= 3` divides the value of *$i* by 3. Note that integer arithmetic is used; thus, if *$a*=5, the command `$a/=3; message "$a"` yields 1, not 1.6667. (Same as `$i = $i / 3`)
`%=`	`$i %= 5` changes the value of *$i* to the remainder of the value of *$i* divided by 5. (Same as `$i = $i % 5`)
`.=`	`$s .= ".pub"` concatenates the string `.pub` to the end of the string represented by *$s*. (Same as `$s = $s . ".pub"`)
`&=`	`$i &= 7` zeros all but the low order 3 bits of the value of *$i*. (Same as `$i = $i & 7`)
`|=`	`$i \|= 1` sets the low order bit of the value of *$i* to 1. (Same as `$i =`

Expression operators assignment table (continued)

	`$i	1`)
`^=`	`$i ^= 1` inverts the low order bit of the value of *$i*. For example, if *$i* is 5, sets *$i* to 4. (Same as `$i = $i ^ 1`)
`<<=`	`$i <<= 2` multiplies the value of *$i* by 4. (Same as `$i = $i << 2`)
`>>=`	`$i >>= 2` divides the value of *$i* by 4. (Same as `$i = $i >> 2`)

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Conditional Expression

`? :`	For the expression `e1 ? e2 : e3`, if the result of the expression `e1` is true (non-zero), the value of the expression `e2` is returned and the expression `e3` is not evaluated. Otherwise, if `e1` is false (zero), the value of `e3` is returned and the expression `e2` is not evaluated.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Logical “or”

`||`	`e1 || e2` returns a value of `1` if either *e1* or *e2* is true; otherwise returns `0` (zero).

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Logical “and”

`&&`	`e1 && e2` returns a value of `1` if both *e1* and *e2* are true; otherwise returns `0` (zero).

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Array Membership

`in`	`$i in $a` returns a value of `1` if *$a* [`$i`] exists, otherwise it returns `0` (zero). (Does not have the side effect of creating *$a [$i]*.) Numeric index sample: `25 in myarray` tests for presence of the 25th index of the array *myarray*. Associative index sample: `'candy' in price` tests for the presence of the `candy` index in the array *price*.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Bitwise “or”

`|`	`$i | $n` returns the bitwise OR of the numbers *$i* and *$n*.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Bitwise “xor”

`^`	`$i ^ $n` returns the bitwise XOR (exclusive or) of the numbers *$i* and *$n*.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Bitwise “and”

`&`	`$i & $n` returns the bitwise AND of the numbers *$i* and *$n*.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Relational

It is important to note that the order in which items appear in this table are arbitrary. All operators have equal precedence to ACL.

Relational table

<	`$x < 5` returns a value of `1` if the value of *$x* is less than 5; otherwise returns `0` (zero).
>	`$x > $y` returns a value of `1` if the value of *$x* is greater than the value of *$y*; otherwise returns `0` (zero).

Relational table (continued)

<=	`$x <= $q` returns a value of `1` if the value of *x* is less than or equal to the value of *$q*; otherwise returns `0` (zero).
>=	`$x >= 3` returns a value of `1` if the value of *$x* is greater than or equal to 3; otherwise returns `0` (zero).
==	`$x == 5` returns a value of `1` if the value of *$x* is equal to 5; otherwise returns `0` (zero).
!=	`$s != "string"` returns a value of `1` if the value of *$s* is not "`string`"; otherwise returns `0` (zero).

The relational operators (<, >, <=, >=, ==, and `!=`) use arithmetic comparisons if both operands are numerals; otherwise, string comparisons are used. White space may delimit operators, but it is not required.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Shift Operators

It is important to note that the order in which items appear in this table are arbitrary. All operators have equal precedence to ACL.

Shift Operators table

<<	`$i << 2` shifts the (unsigned) value of *$i* left by 2 bit positions.
>>	`$i >> 2` shifts the (unsigned) value of *$i* right by 2 bit positions.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Addition, Subtraction, String Concatenation

Note

The order in which items appear in this table is arbitrary. All operators have equal precedence in ACL.

Addition, subtraction, string concatenation table

+	`$n + $m` evaluates to the sum of *$n* added to *$m*.
–	`$i – 2` evaluates to the value of *$i* less 2.
.	`"ch1" . ".xml"` evaluates to "`ch1.xml`".

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Multiplication, Division, Remainder

The order in which items appear in this table are arbitrary. All operators have equal precedence to ACL.

Multiplication, division, remainder table

*	`$n * $m` evaluates to the product of *$n* times *$m*.
/	`$i / 9` evaluates to the value of *$i* divided by 9. Note that integer arithmetic is used; thus, if `$len=30`, the command `$divvy=$len/9; message "$divvy"` yields 3, not 3.3334.
%	`5 % 2` evaluates to the remainder of 5 divided by 2, or "`1`".

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Unary Minus

–	`–4` evaluates to the negative number, "`–4`".

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Logical Negation

!	`! $e` returns a value of `1` if the value of *$e* is zero or null, indicating it is true. Otherwise returns `0` (zero), indicating it is not true.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Bitwise Negation

~	`$n=~7` returns the bitwise NOT of its operand, in this case ~7 = –8.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Increment

++	`$i = ++$n` (prefix form) first increases the value of *$n* by 1, then sets *$i* to this new value. Alternately, `$i = $n ++` first sets *$i* to the value of *$n*, then increases the value of *$n* by 1. (*$i* must be either a scalar variable or an array reference, as must *$n*.)

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Decrement

--	`$i = -- $n` (prefix form) first decreases the value of *$n* by 1, then sets *$i* to this new value. Alternately, `$i = $n --` first sets *$i* to the value of *$n*, then decreases the value of *$n* by 1. (*$i* must be either a scalar variable or an array reference, as must *$n*.)

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Grouping

( )	`4 * (3 + 2)` evaluates the sum of (3 + 2) before multiplying by 4; thus the string would evaluate to 20 rather than to 14.

The following operators are numeric and require numeric operands:
`+, -, *, /, &, ^, |, <<, >>, %, ~, ++, and --`

Logical Expressions

Logical expressions are those using the operators `!` (for logical negation), `&&` (the "and" operator), or `||` (the "or" operator). In logical expressions, a string that is not the null string or "`0`" (zero) is evaluated as true. The result of a logical operator is `1` for true and `0` for false. Short circuit expression evaluation is done for the "and" and "or" operators. This means that with the "and" operator (`&&`),

evaluation ceases as soon as a false is found, for the expression must be false. Likewise, with the “or” operator (| |), evaluation ceases as soon as a true is found, for the expression must be true.

Operands

Operands for expressions are any of the following:

Operands: number

number	A string of decimal digits 0 to 9. A sequence of digits starting with 0 is treated as an octal integer (base 8). A sequence of digits beginning with 0x or 0X is taken as a hexadecimal number (base 16). Hexadecimal digits include a (or A) through f (or F) with decimal values 10 through 15. For example, the number 31 can be written as 037 in base 8 or 0x1f in base 16.

Operands: string

"string"	(Note double quotation marks.) A string in which variable substitution may be done. The usual C-style backslash sequences are recognized in double quoted string literals: \b backspace \f formfeed \n newline \r carriage return \t tab \ddd octal value ddd, where ddd is 1 to 3 digits between 0 and 7 \c any other character taken literally, for example, \" for ", \ ` for ` and \\ for \
'string'	(Note single right quotation marks.) An arbitrary string which is uninterpreted—that is, variables are not substituted.
`string`	(Note single left quotation marks, or back quotes.) A string interpreted as a shell command, and the output of that command (or pipeline) is the value of the operand, as in the Bourne or C-shells. White space in

Operands: string (continued)

	the output (space, tab, and newline) is replaced by single blanks between words. Leading and trailing white space is removed. Variable substitution is performed on the string before it is sent to the shell. The command is executed each time the expression is evaluated.
$name	A variable. Braces may be used to delimit the name from following text when used in a quoted string, (for example, "${tagname}_help"). Note: The dollar sign is not required for variable references inside functions or in scopes outside the main package.
$array [e1]	An array reference. The string value of the expression e1 is used as the array subscript.
name (e1, e2 , . . .)	A built-in function that takes arguments that are expressions.

(*name) (a1, a2 , . . .)	Is an indirect function call. It represents a call to a function whose name is the value of the variable name, which must be a lvalue, that is, either a scalar variable or array element. The function name is looked up in the current package if no package qualifier is given. name may specify either a user-defined or a built-in function. As an example, here is a function that calls another function whose name is given as a parameter:
	`function callback(func, data) {if (defined("$func()")){ (*func)(data)}}`
	The call to `defined` verifies that the function is defined, since otherwise the script will be aborted if *func* specifies an unrecognized function name. The indirect function operator is more efficient than using the `eval` function to build a call to a function. For example:
	`eval(func ."(data)")`
	since it doesn't have to compile an expression every time the function is executed.

Leading white space in a string is ignored in cases where the string is interpreted as a number.

Arbortext Editor uses the dot (.) as a concatenation operator, which allows you to use built-in functions, such as `substr`, to build strings. This means that for a document named `Ch2`, an operand such as `${docname}.bak` would be interpreted as `Ch2bak`, and not as `Ch2.bak`. In cases where this is not desired, you must use quotation marks to indicate that the dot is to be taken literally (for example, `"${docname}.bak"`).

Using Conditional Logic

The Command Language has four commands available to conditionalize execution of commands to a specific condition. These commands are `if`, `while`, `for`, and `switch`.

The `if`, `while`, `for`, and `switch` commands do not change the $status variable (a predefined Arbortext Editor variable), unless a syntax error results during evaluation of the logical expression describing the conditions in which the commands are to be executed (*expr* in the syntax statement above). In this case, $status is set to 2 and the statement is aborted.

If you are typing an `if`, `while`, `for`, and `switch` command at the command line, the entire command must appear on one line. However, if the command is read from a command file, the block of commands may extend across multiple lines. Line breaks can occur at command boundaries or, for a conditional command like `if`, `while`, `for`, and `switch` at the start of the command block, that is, before or after the left brace ({). If you need to break the line in the middle of a command, put a backslash at the end of the line preceding the break.

if, while, for, switch

The Command Language has four commands available to conditionalize execution of commands to a specific condition. These commands are `if`, `while`, `for`, and `switch`.

The if statement has the following structure:

```
if (condition) {
 commands
} else {
 commands
}
```

If the condition defined by *condition* is true (that is, not equal to 0 (zero)), then the first group of commands is executed. If the condition is false (that is, equal to 0), then the second group of commands (defined by the `else` statement) is executed. The `else` statement in this command is optional.

In the following example, a `find` command is executed. The if statement checks the $status variable to evaluate if the find was successful. If the condition $status == 0 (that is, if the find was successful, and thus the status variable is equal to 0), then the string is replaced, otherwise a message is issued indicating that the string was not found.

```
find "Peter"
if ($status == 0) {
 delete_mark
 insert_string "John"
} else {
 message "Peter not found!"
}
```

The `while` command described below is a loop statement that relates repeated execution of a group of commands to a specific condition.

```
while (condition) {
 commands
}
```

The commands defined by *commands* are repeatedly executed until *condition* becomes false (that is, equal to 0 (zero)).

In the next example, the `find` and `replace` commands are repeatedly executed until the `find` command is unsuccessful, which sets the status variable to something other than `0` (zero). The result is that every occurrence of the string in a document is replaced, rather than only the first one, as in the previous example.

```
find "Peter"
while ($status == 0) {
 delete_mark
 insert_string "John"
 find "Peter"
}
```

The `for` command can also execute a group of commands repeatedly. There are two ways to use the command.

In the following example, for is used to execute a group of commands for every element in an array. If $array is an array variable with five elements, then *commands* is executed five times (for each element in this array). Each time, $index has the current value of the array index.

```
for ($index in $array) { commands }
```

The second way to use the `for` command is to relate its execution to a counter. In the following example, $count is set to a start value of `1`. The commands are executed while $count is less than `5`. After every cycle the counter is incremented by `1`. This means that the commands are executed exactly four times.

```
for ($count=1; $count<5; $count+=1)
{
commands
}
```

Note that at the exit of the `for` loop the value of $count is equal to `5`.

The syntax of the second use of `for` is just like in C language:

```
for (expression1;
expression2;expression3)
{
commands
}
```

where each expression is optional. This form is equivalent to:

```
expression1
while (expression2) {
 commands
 expression3;
}
```

The `switch` command selects from a group of statements based on the value of the expression. The form of the statement is similar to that of the C programming language. It has the following syntax:

```
switch (expression)
 {
 case constant1:
 statements
 break
 case constant2:
 statements
 break
 …
default:
 statements
 break
}
```

The body of the switch statement must be enclosed in braces. The case statements do not need to be enclosed in braces.

The case and default prefixes do not alter flow of control. The break statement may be used to transfer control out of the switch statement.

Unlike C, the case constant value does not only need to be an integer. The constant may be a string value delimited by single or double quotes, or a regular expression delimited by slashes. For example, this:

```
switch (x) {
case 1:
 message "x is 1"
 break
case "one":
 message "x is 'one'"
 break
case /this|that/:
 message "x contains 'this' or 'that'"
 break
}
```

is equivalent to this:

```
 if (x == 1) {
 message "x is 1"
} else if (x == "one") {
 message "x is 'one'"
} else if (match(x, "this|that")) {
 message "x contains 'this' or 'that'"
}
```

Since the `match` function is used to evaluate a case prefix that specifies a regular expression, the statements following the case may use the match functions `match_result`, `match_start`, and `match_length` to extract parts of the matched expression.

If the case constants are all integer values or string constants of a single character, a jump table may be used to transfer control to the matched value. This executes considerably faster than a series of `if, else if` clauses. If the case values are all string values, a binary search is used (for more than three cases). A series of `if, else if` statements are compiled if the case values are regular expressions or are a mixture of string and integer value.

The case constant label allows for variable substitution. For example:

```
readIncompete=-2
readError=-1
readEOF=0
switch (read(ch, buf, 512)) {
case $readIncomplete:
 ...
case $readError:
 ...
case $readEOF:
 ...
default:
 ...
}
```

break and continue

The `break` and `continue` commands are useful in `if`, `while`, and `for` loops. The `break` command is typically used in combination with a condition. If a `break` command is executed in a loop, it immediately forces the loop to be broken off.

In the following example, the `while` loop is broken off when a `noname` tag occurs to the left of the string Peter. In that case, the `while` loop is broken off, the replacement does not take place, and the search terminates.

```
find "Peter"
while ($status == 0) {
 if ($tagname == "noname") {
 message "noname tag encountered!";
```

```
 break
}
 delete_mark
 insert_string "John"
 find "Peter"
}
```

The `continue` command also breaks off the execution of the loop but does not exit it. It causes the rest of the statements in the current cycle of the loop to be skipped and forces the loop to start at the beginning of the next cycle.

In the following example, the `break` is replaced by a `continue`. As in the above example, if the string Peter is preceded by a `noname` tag, it is not replaced. However, unlike the previous example the loop continues its search. It cycles between `continue` and `while`, since that status is still true, thus it is not broken off.

```
find "Peter"
while ($status == 0)
{
 if ($tagname == "noname")
 {
 message "noname tag encountered!"
 continue
}
 delete_mark
 insert_string "John"
 find "Peter"
}
```

The `break` command can also be used within a `switch` statement to transfer control after the end of the `switch` body. For example:

```
switch (str) {
case |x|:
 if (str == "x1") {
 break
}
case |y|:
 message "$str starts with y"
 break
}
```

Using Looping and Conditionals

Until you become familiar with the syntax and power of logical expressions and variables, you may find it difficult to write `if/else`, `while`, and `for` commands.

The Looping and Conditionals help book contains examples of if/else, while, and for commands. By examining these examples, you will get a sense of how expressions and variables are used to build these commands.

Start by looking over the syntax statements for the `if/else`, `while`, and `for` commands. Then view the online help for variables and expressions . Finally, work through the examples in the Looping and Conditionals help.

Example: Test a Document Type

The following command notifies you if the document you are editing is a plain ASCII file rather than an SGML document:

```
if ($doctype=="ascii") {
 message "Notice: The file you are editing \
 is not an SGML document. You can still use \
 Arbortext Editor to edit this file but \
 you cannot enter tags.";}
```

The logical expression `$doctype=="ascii"` tests the value of the variable *$doctype*. If the value of this variable is `ascii` (that is to say, if you are editing an ASCII document), Arbortext Editor prints the warning message expressed by the `message` command.

You could put this in the document *`docname`*`.acl` file, but then you would not get the message if you switched documents with the `edit` command. To get around this, you could put the above `if/else` command in a command file called `asciiwri.acl`. Then put the following commands in your *`docname`*`.acl` file:

```
source asciiwri.acl
alias pub {edit $1; source asciiwri.acl;}
```

If you use the `pub` command to switch files instead of the `edit` command, you are notified if you have switched to an ASCII file.

Example: Determine a File's Write Status

The following `if` command, which would presumably be embedded in a procedure that defined the variable *$file*, uses the `access` built-in function to determine the write status of a file:

```
if ( access($file, "w" ) ) {
 message "can't write file $file"
}
```

Example: Test the Location of a Document

These commands modify the running headers for any memo documents in a particular directory:

```
$workingdir=`pwd`
if (index($workingdir,'conf')!=0 && \
```

```
 $doctype=="memo") {modify_tag -global document \
 scilevel=secret;}
```

`pwd` is used to print the complete path for the working directory. The path is then used as the value of the newly defined variable *$workingdir*. Statements of the format `$`*`xxx`* = are used to assign values to variables or to create variables and assign them values. The name of the variable appears to the left of the = sign and the value of variable appears to the right. `pwd` is a command that prints the complete path for the working directory. You can pass commands from Arbortext Editor to the operating system by enclosing them in left quotes.

The condition for this `if` command consists of a complex expression made of two smaller expressions.

The first of the smaller expressions:
`index($workingdir,'conf')!=0`

uses the `index` function to test whether the path for the working directory includes the string "`conf`" for confidential (presumably, any memos in the directory `conf` or a subdirectory of that directory should have the special header). The `!=` (NOT) operator compares the result of the index function with `0`. If the value is not `0` (that is, the string "`confidential`" appears in the path) then the first expression is true. If it is `0`, the expression is false.

The second of the smaller expressions:
`$doctype==`"`memo`"

tests to see whether the value of the variable *$doctype* is "`memo`". In this case, if the value is `0` (the value for *$doctype* was "`memo`"), the expression is true. If the value is not `0`, the test is false.

Finally, the AND operator, `&&`, compares the results of the two tests. For the Arbortext Editor `modify_tag` command to be executed, both expressions must be true. (The `modify_tag` command changes the security level, or "`scilevel`", to "`secret`") If the document is not a memo or is not in the directory `conf`, the security level is not affected.

A Further Modification

You may want to add another `if/else` command to the commands shown in the Test the location of a document example. Using the `pwd` command to print the working directory slows things down a little bit. In some cases, the value of the variable $dirname is already the complete path. You can speed things up in some cases by checking to see whether the value of $dirname is already a complete path and if it is, using it as the value for $workingdir:

```
if (substr($dirname,1,1)=='\') {
 $workingdir=$dirname;
} else { $workingdir= `pwd`;}
 if (index($wd,'conf')!=0 && $doctype==memo) {
 modify_tag -global document \
```

```
 scilevel=secret;}
```

$dirname is a predefined Arbortext Editor variable. Its value is the name of the current document directory. The initial `if/else` command uses the `substr` function to find out whether the directory name begins with a backslash (\). If the name does begin with a backslash, the value of $dirname is a complete path, in which case it can be used as the value for $workingdir. If the name does not begin with a slash, the status of the `index` function will be 0, the initial `if` condition will not be true, and the `else` clause is invoked, calling `pwd` to get the complete path.

Example: Substituting Tags

Currently, you cannot use the `substitute` command to insert, delete, or change tags. The following series of commands (which would be put in a file and called with the `source` command) gets around this restriction by interactively inserting a tag pair around every occurrence of the phrase selected. By substituting `change_tag` or `delete_tag` for `insert_tag` in the command sequence that follows, you could design a similar procedure which would delete or change tags in the highlighted phrase.

```
set sgmlselection=off
$phrase=$selection
set sgmlselection=on wrapscan=off
if ($tagname != "") {
 if (substr($tagname,1,1) == '/') {
 $TAG=substr($tagname,2,length($tagname)-1);
}
 else {
 $TAG=$tagname
}
} else {
 $TAG='NO_TAGS'
}
if ( "$TAG" == "NO_TAGS" ) {
 message "No tags allowed in ascii file"
} else if ("$phrase" && "$TAG") {
 find -sel "$phrase";
 while ($status == 0) {
 $ack=response("Change?", "Yes", "No", "Cancel")
 if ($status != 0 || $ack == 3) { break;}
 if ($ack == 1) {
 break;
}
 if ($ack == 1) {
 insert_tag $TAG}
}
 find
}
} else {
```

```
 message "You must select something first"
}
unsetvar TAG phrase
```

For example, if you want to put emphasis tags around almost every occurrence of “Arbortext Editor” in your document, find the first occurrence of the phrase in your document and put emphasis tags around it. Then select the entire phrase (including the emphasis tags) and source the file containing the commands listed above. Arbortext Editor then finds each occurrence of the phrase and prompts you to insert the tag pair. To insert the tag, click on the **Yes** button on the prompt panel. To skip the phrase (for example, if the occurrence were already tagged), click on the **No** button. To get out of the search-and-tag operation (abort the loop), click on the **Cancel** button.

This command script could also be written as a user-defined function.

Example: Tailoring Dash Insertion

Arbortext Editor cycles through three different types of dashes (- or hyphen, – or en dash, and — or em dash) when you press the - key. The following commands change the order of this cycling.

```
# Here we define individual aliases to insert each
# kind of dash:
#
alias insert_hyphen
 {
 insert_string -sgml "-X";
 delete_character;
}
alias insert_ndash {insert_string -sgml "&ndash;"}
alias insert_mdash {insert_string -sgml "&mdash;"}
#
# Here we define what dash type should be inserted
# for each case. To change the order, only these
# aliases need to be swapped:
#
alias insert_dash_after_non_dash insert_hyphen
alias insert_dash_after_hyphen insert_ndash
alias insert_dash_after_ndash insert_mdash
alias insert_dash_after_mdash insert_hyphen
#
# Insert the appropriate dash depending on what's
# already there.
#
alias insert_dash
 {
# assume pending delete, so if there is a selection,
# delete it before inserting the dash
 if ( selected() > 0 )
```

```
 {delete_mark;}
# now select the character before the cursor
clear_mark;
mark begin;
caret 0,-1;
mark end;
# figure out what dash to insert based on the previous
# character, since sgmlselection may be set on or off,
# we need to test two cases where the SGML and ASCII
# versions are different.
if ( $selection == "-" ) # type one dash
 {
 delete_mark;
 insert_dash_after_hyphen;
}
else if ( $selection == "&ndash;" || \
$selection == "--" ) # type two dashes
 {
 delete_mark;
 insert_dash_after_ndash;
}
else if ( $selection == "&mdash;" || \
$selection == "---" ) # type three dashes
 {
 delete_mark;
 insert_dash_after_mdash;
}
else
 {
 clear_mark;
 caret 0,+1;
insert_dash_after_non_dash;
}
};
#
# Finally, we map the hyphen/dash/minus key to our
# insert_dash alias.
 map edit - insert_dash;
```

If you were operating in an environment when pending delete was disabled, you would want to remove the lines that deleted a selected region.

ACL Functions Defined

Functions are different than commands because they return a value that can be assigned to a variable. A function takes arguments that are specified in parentheses directly following the function name. Here is the syntax for a function:

```
$value = functionname(
argument1, argument2, . . .)
```

Assigning the value to a variable is not mandatory.

Functions as Expressions and Commands

Except as otherwise noted, all arguments may be expressions. Also, the ACL parser recognizes function calls in the context of a command. This means that, in addition to using a variable assignment or logical expression to call a function (such as `$n = split("a b c", $arr, " ")`, you can type a command that contains a function call at the command line. Here is an example:

```
split("a b c", $arr, " ")
```

When you call a function in this manner, the value returned by the function is discarded.

Channel Functions

The channel functions provide interfaces to the Berkeley socket library, which allows network applications to be written. The `open` functions return channel identifiers that may also be used with most file identifier functions (for example, `read`, `write`, and `close`).

Several `open_` routines return a channel identifier that may be used by subsequent I/O functions. A channel identifier is a small integer file identifier that is returned by `open` and can be used with the file identifier routines `getline`, `put`, `close`, `read`, and `write`. These channel functions allow binary data to be transmitted.

Object Identifier (OID) Functions

In addition to strings and integers, there is another data type in ACL called an object identifier (OID).

An OID is a unique identifier that locates an element within the structure of a parsed document instance. The OID also records to which document instance the element belongs. This means you can save an OID in a variable, and then later refer to it in OID functions. It makes no difference what the current document is.

An OID is assigned to each start tag when it is parsed or inserted into the document instance. An OID is valid for the duration of an editing session or until the associated object is deleted, for example, by a cut operation.

Not all SGML markup within Arbortext Editor are given OIDs. In particular, character entity references represented by internal tag names starting with `&`, and the `_include` and `exclude` internal tags used to represent included marked

sections, are not assigned OIDs because they are not part of the element structure. End tags are not assigned OIDs because the OID for the start tag is sufficient to locate the end of the structure.

File and text entity tags do have object identifiers. That is, OID functions such as `oid_next` and `oid_child` return OIDs for file and text entity tags. These entity objects appear as elements with no content, and `oid_empty` always returns the value of `1`.

To descend into an entity, the `oid_entity_first` function may be used to return the top-level element contained within the entity. For external file entities, the OID returned will be in a different document tree. If *oid* is the OID of a file entity, `oid_doc` (*oid*) != oid_doc(oid_entity_first(*oid*)) For text entities, the OID returned is in the same tree. Related functions are `entity_first`, `entity_doc`, and `entity_name`.

When the `viewchangetracking` on page 923 option is set to **all** the oid-walking functions will "see" and be able to return change tracking markup. Under any other circumstances they will not. It is acceptable to pass a change tracking markup oid to an oid function no matter what the view setting.

There are several functions that operate on OIDs to permit efficient navigation and interrogation of the document structure. In addition, the logical operators and may be used to compare two OIDs for equality. The logical operators , , , and test the ordering of two OIDs. For example, is true if oid1 occurs before oid2 in the document instance. The relational operators do not test structural relationships; the `oid_level` function can be used for testing if one OID contains another.

The only other operator that permits OIDs as operands is the assignment operator. It is incorrect to use an OID with any other operator (for example,) or to compare an OID with a number or string.

User-Defined Functions

You can define your own functions using this form:

```
function name(parameter-list)
{
commands
}
```

parameter-list can contain up to 50 scalar and array variables that are separated by commas. You declare an array variable by appending brackets ([]) to the variable name.

When a function is called, scalar variables are passed by value, and array arguments are passed by reference. You can force a scalar variable to be passed by reference by adding an ampersand (&) to the variable name. Take this function as an example:

```
function f(a, &b, c[]) {}
```

In the example above, the scalar variable "a" is passed by value. The scalar variable "b" and the array variable "c" are passed by reference.

As is the case in C++, you can give trailing arguments default values. Here is an example:

```
function f(n, m=10) {}
```

This example declares a function that may be called with one or two arguments. If only one argument is passed, the value of "m" is 10. The default value is an expression that's evaluated at the time of the call. Any variables that are referenced are in the scope of the caller.

The names specified in *parameter-list* are local to the function. All other variables are global. You can introduce additional local variables by using the `local` command, as shown in this example:

```
local x, y=10, z[]
```

This example declares scalar variables x and y and array variable z to be local to the current block, which ends with a right bracket (]). Scalar variables can be given initial values. In the example above, `y` has a value of `10`. The `local` command is an executable command, so default values are evaluated each time the command is executed. This also means it's not efficient to use `local` commands inside loops that are executed many times.

In the C programming language, you can declare a local variable that's already defined at the current level. You cannot do this in ACL. The following example illustrates this point:

```
function f(x)
{
 # illegal, hides parameter
 local x;
 local m=1
 {
 # ok
 local m=2;
 # ok
 local x[];
}
 # illegal, hides previous value
 local m;
}
```

In ACL, the `local` command is only allowed inside function definitions.

A function may return a value if the `return` command, which takes an expression as an optional argument, is used with the function. For example, `function fac(n) { return n > 1 ? n * fac(n-1) : 1}` is the most common factorial function. If no value is given for the `return` command, or the function fails to execute a `return` command, the result of the function is undefined. It's an error to refer to the result of the function in this case.

Note

The `return` command is also valid inside aliases and source files. In this case, the argument (if any) indicates whether or not the alias or source file succeeds: zero means success, while nonzero means failure.

A function definition must appear at the outermost scope level. That is, you can't define a function inside a function, an alias, or a block. However, it is possible to define a function dynamically using the built-in `execute` function.

Functions must be defined before they are called. The `function` command defines the function name after the *name* argument is parsed. This allows recursive functions to be declared. You can declare a forward reference using this form:

```
function name () {}
```

Unlike aliases, the text of the function is not kept in memory, so there is no limit on the size of the function body.

Within a function definition, a variable term in an expression need not start with a dollar sign ($). Here is an example:

```
function backup_and_save()
{
 local bakfile = filename. ".bak"
 copy_file $filename $bakfile
 save
}
```

However, the dollar sign is still needed to introduce variable substitution in commands that do not take expressions. The dollar sign is also required in double-quoted strings within expressions.

Also, unlike aliases, you don't have to add `execute` to commands in which variable substitution is done at parse time. In the example above, an implicit `execute` is added to the `copy_file` command, so the values of the variables are not used until run-time.

Packages

Packages let you declare global variables and functions whose name space is localized to one or more command files. You declare a package using the `package` command on page 689 at the beginning of a file. Here is an example:

```
package myapp
```

This package defines a new symbol table in which all global variables and functions in the remainder of the file are placed. You can refer to variables and functions in other packages by adding them to the package name with two colons (::). Here are some examples:

```
main::selection, history::set_cmd_line(), myapp::myvar
```

Predefined variables such as `doctype`, `status`, `selection`, etc., are defined in the package `main`. If you want to reference these predefined variables and functions inside a new package, the new package must be preceded by the `main` package, such as:

```
main::status
```

In packages other than `main`, it's not necessary for a dollar sign ($) to precede a variable term inside an expression.

All function names defined in a package are local to the package. It is possible, however, to "export" the function name to the main package (or another package) by using a package prefix on the function definition. Here is an example:

```
function main::fac(n) { ...}
```

Note

It's best not to define functions (and aliases) that will be exported to the `main` package because this could conflict with Arbortext Editor internal functions or with new built-in functions provided in later Arbortext Editor releases.

The example shown above puts the function name `fac` into the main package, but any variables in the function body are still local to the current package.

The package declaration may also be used within a function definition to switch the current package. The scope of the package declaration is the same as the `local` command. That is, the previous package is restored when the end of the block is reached, which is indicated by a right curly bracket (}), or another `package` command is executed. Here is an example:

```
package myapp
status=0
function f(x)
{
 # refers to myapp::status
 status = 1;
 {
 package main
 # refers to main::status
 status = 0;
}
 # refers to myapp::status
 status = 2;
}
```

If you're in a package other than `main`, and that package calls out functions (including built-in functions) defined in `main`, it's not necessary to precede those functions with the `main` package prefix. When a function call without a package prefix is compiled, the parser looks up the name of the function in the active function list. If the function name isn't found, the parser tries to determine if the function is a user-defined or built-in function in the `main` package.

Unlike package `main`, a reference to an undefined variable does not refer to an environment variable of the same name. For example, in the `main` package, *$PATH* normally refers to the path environment variable. In any other package, `$PATH` refers to a package local variable.

The name space for aliases is not affected by packages. Aliases are always compiled as if they had been defined in the `main` package. Like the `local` command, the `package` command is not valid in an alias. To refer to variables (and functions) that aren't in the `main` package, you must use the package name prefix within an alias.

Built-in Arbortext Editor commands may also be prefixed with the main:: qualifier. This allows you to define an alias that unambiguously refers to a primitive command. Here is an example:

```
alias myquit {
 # do something
 main::quit; # really quit
}
```

Without the prefix, the reference to `quit` might actually invoke some alias of quit.

The `package` function on page 480 can be used to change the current package to a package previously specified by the `package` command.

Callback Functions Introduction

A callback function is a user-defined function that is called during specific events in Arbortext Editor and allows you to customize editing operations. Callback functions provide both a global and a local level of registration or notification to isolate events, while hooks provide only global effect (unless additional user functions are supplied). For example, you might want a particular function called whenever a specific document is saved or a help window has a quit request.

Note

Using Callbacks instead of Hooks

Where possible, use the callback functions instead of hook functions. In some cases, disabling and resetting hooks using `set` *hookname* may disrupt the functioning of Arbortext Editor applications.

Refer to Callbacks on page 975 for a reference of available ACL callbacks. Refer to Hooks on page 937 for a reference of ACL hooks.

At the session level, there are two basic callback functions. The `session_add_callback` function adds a specified function as a callback that's invoked whenever the Arbortext Editor session occurs. The `session_remove_callback` function removes the function as a callback for session monitoring.

At the window level, there are also two callback functions. The `window_add_callback` function adds a specified function as a callback to that is invoked whenever the specified action occurs in the specified window or in any window if *win* is 0. The `window_remove_callback` function removes the function as a callback.

At the document level, there are two basic callback functions. The `doc_add_callback` function adds a specified function as a callback that is invoked whenever the specified action occurs in the specified document, or in any document if *doc* is 0. The `doc_remove_callback` function removes the function as a callback.

At the dialog item level, there are two basic callback functions. The `dlgitem_add_callback` function adds a specified function as a callback that is invoked whenever the specified action occurs at the specified dialog item. The `dlgitem_remove_callback` function removes the function as a callback.

You can also invoke callbacks after a specified amount of time. The `timer_add_callback` function adds a specified function as a callback that is invoked after a specified interval. The `timer_remove_callback` function removes the function as a callback.

The basic structure of a callback is (document-level callbacks are used for these examples):

```
doc_add_callback (doc, cbtype, funcname)
```

The function specified by *funcname* is a callback that is invoked whenever the type of action specified by *cbtype* occurs in document ID *doc*.

doc is either a valid document ID, or 0. If *doc* is 0, the callback applies to all documents.

cbtype is a string and must be a callback.

funcname should be a fully-qualified function name with no argument list. As examples of valid and invalid names for *funcname*:

Callback function specification

Callback function name	Description
`'mycallbacks::mycback'`	Valid form, where mycallbacks is the package name
`'mycback'`	Valid, but specify the package name to avoid ambiguity
`'xxx::mycback()'`	Invalid. The callback function name can't include parentheses.

In the next example, the function *mycallback* is removed as a callback of type *cbtype* on the document *doc*.

```
doc_remove_callback( doc, cbtype, mycallback)
```

An example of a function that is notified whenever a cut is requested on docs 5, 7 or 10:

```
package mycallbacks
function cut_callback( doc, op )
{
 if ( op == 1 )
 {
 # no reason not to proceed
 return 0
}
 else if ( op == 2 )
 {
 # this is where processing occurs
 # we want to block the user from
 # cutting anywhere inside <para> tags in this doc
 if ( inside_tag('para', doc) )
 {
 response( 'Can\'t cut inside <para> tags.' )
 return -1
}
 else
 {
 return 0
}
}
}
...
doc_add_callback( 5, 'cut', 'mycallbacks::cut_callback' )
doc_add_callback( 7, 'cut', 'mycallbacks::cut_callback' )
doc_add_callback( 10, 'cut', 'mycallbacks::cut_callback')
```

Whether or not a callback function is called with any parameters depends on the particular function. It is an error to assign a callback function when its parameters do not match (although this is not detected until the function is called).

Callback functions that take `op` as their last argument are called twice in succession. The first call (`op=1`) checks for approval to act. A return value of `0` means continue execution. A return value of `-1` from any callback means stop the operation without calling any more callbacks that may be registered, do not make any `op=2` callback calls, and skip basic Arbortext Editor processing. The second call (`op=2`) performs the action unless the operation has been stopped earlier. A return value of `0` allows basic Arbortext Editor processing after all callbacks have been called. A return code of `-1` from any callback prevents basic Arbortext Editor processing.

Hook Functions Introduction

A hook is a user-defined function that is called at strategic places by Arbortext Editor so you can customize editing operations. Setting a hook has global effect unless you build in additional user functions to specify a local level.

Note

Callback functions provide an easier interface than direct hooks. For instance, direct hooks only provide global monitoring, whereas callbacks can be global or specific to a document or a window. Therefore, callbacks are the preferred method for custom processing functions when both a hook and a callback are available.

Refer to Hooks on page 937 for a reference of available ACL hooks. Refer to Callbacks on page 975 for a reference of ACL callbacks.

The built-in functions `add_hook` and `remove_hook` are convenience routines that enable and disable hook functions. The `add_hook` function has the form:
`add_hook(`*`hookname, funcname`*`[, `*`prepend`*`])`

where *hookname* is the name of the hook `set` option, *funcname* is a string specifying the name of the user-defined function to call, and *prepend* is an optional flag which if non-zero specifies the function should be added to the head of the hook function list. `add_hook` will not add *funcname* if it is already present in the list. The `remove_hook` function has the form:
`remove_hook(`*`hookname, funcname`*`)`

where the parameters *hookname* and *funcname* are the same as for `add_hook`.

Whether or not a hook function is called with any parameters depends on the particular hook. It is an error to assign a hook function with mismatched parameters (although this is not detected until the hook function is called). Some

hook functions must return a result. For hooks that do return a result, `-1` is generally a special return value. This value means the edit operation was completed by the callback, no additional callbacks are needed, and Arbortext Editor need not continue processing the command.

A hook function may change the associated `set` option (by using `add_hook` or `remove_hook`), although the change will not take effect until all functions on the current hook list have been called. However, any hook may still return a value of `-1` to abort the processing of the remaining hooks.

When a hook function is called, the hook is suspended so that the function will not be reentered if it executes a command or function that would result in the hook function being called again.

Formatting Pass Reduction in ACL

The term "pass reduction" refers to minimizing the number of formatting passes Arbortext Editor takes when formatting a document instance for preview or printing. Without pass reduction, documents that refer to page numbers or page locators (for example, in a table of contents, in cross references, or having the final page number in the header or footer), will require two passes the first time they are formatted during an editing session. With pass reduction, many documents can be formatted satisfactorily with one pass. This approximately halves the time required for formatting, at the risk of some loss in typesetting precision in the form of extra white space adjacent to page numbers.

The basic mechanism for pass reduction is a process where the resolved page locators are overlaid in space reserved for them during the initial pass. Whether or not there is visible extra space, and whether it degrades a document's appearance depends on the FOSI, and the document instance.

Arbortext Editor provides users with access to pass reduction controls with two preferences. For ACL scripting purposes, these preferences are available in the form of two set commands.

The set commands are `set overlaypagenumbers` on page 858 `set overlayunderflowtolerance` on page 859.

The `set overlaypagenumbers` command enables/disables pass reduction (the default is enabled). The `set overlayunderflowtolerance` command sets the amount of disparity to accept when a final page variable value is smaller than the space it is overlaid in. Arbortext Editor never accepts an overlay if the new value is larger than the space it is overlaid on (the default is one character).

The related environment variable, *APTNOOVERLAYPAGENUMBERS*, is provided for customers who absolutely do not want page number overlaying to be done, they can totally disable overlaying by setting the environment variable *APTNOOVERLAYPAGENUMBERS* to `Yes`.

The paragraphs below explain how these settings interact with formatting command options. First some terms need to be explained:

- overlaying fails means that it is impossible to overlay page numbers because the change in a page variable affects the structure of the formatted output, not just a character string. This is the case when
  - an index contains entries with multiple page references these can't be overlayed because they may have been merged, or
  - a page variable contains constructs such as elements, pseudo-elements, kerns, that is anything other than a counter value or character string
- overlaying was poor means:
  - an overlaid value was more characters than the original, or
  - an overlaid value was less than the original value by more than overlayunderflowtolerance characters
- overlaying works well means neither of the above conditions was true
- formatting command means any of the `preview`, `format`, or `print composed` commands.

If pass reduction is enabled, and you execute a formatting command with the `onepass` option, the Arbortext Editor formatter makes one pass, and at the end of the pass overlays page numbers so that the output will show resolved page numbers. If overlaying fails or is poor, then the document will be marked as `refmt-needed` and the next formatting command the user enters will cause another formatting pass, and the correct values will be plugged in without being overlaid. If overlaying works well, then the document will not be marked as `refmt-needed`. If desired, the user could force an additional formatting pass using the `force` option with the formatting command in order to obtain a non-overlaid result.

If pass reduction is enabled, and you execute a formatting command with the `allpasses` option, the Arbortext Editor formatter first makes one pass and at the end of the pass overlays page numbers. If overlaying fails or is poor, then the formatter automatically does another pass. If overlaying works well, then the single pass suffices. Again, if desired, the user could force an additional formatting pass using the `force` option with the formatting command in order to obtain a non-overlaid result.

If pass reduction is disabled, and you execute a formatting command with the `onepass` option, then the Arbortext Editor formatter makes one pass, and at the end of the pass, if there are any page numbers that are incorrect, it overlays page numbers so that the output will show resolved page numbers. That part is the same as if pass reduction is enabled. What is different is that the document will be marked as `refmt-needed` if any page number overlaying was done, regardless of whether the overlaying worked well.

If pass reduction is disabled, and you execute a formatting command with the `allpasses` option, the Arbortext Editor formatter makes one pass, and at the end of the pass, if there are any page numbers that are incorrect, a second pass is automatically initiated.

A compromise for users who want highest publishing quality but want the speed of overlaying would be to specify the underflow tolerance as `0`. Then overlaying will be attempted, but will only be accepted for final output if the number of characters matches exactly. How often this happens will depend on the work habits of your authors.

ACL Coding for Better Generated Text Performance

For a series of complex editing operations the ACL developer should consider temporarily disabling generated text updates before performing the operations and then restoring the previous generated text state when finished. This will maximize performance if `gentext` is set to `on` and `gentextautoupdate` is set to something other than `none`, as the gentext will only be updated once when all editing operations are complete, instead of incrementally after each edit.

```
function complex_edit() {
 local previous_gentext_state = option('gentextautoupdate');
 set gentextautoupdate=none;
 [...two or more really complex editing functions here...]
 set gentextautoupdate=$previous_gentext_state;}
```

Note

Turning off screen updates with `doc_update_display(doc, 0)` will also disable generated text processing.

DDE Support

Arbortext Editor users can use Dynamic Data Exchange (DDE) to execute ACL commands and functions within a Arbortext Editor session. This DDE support is intended for integrators who wish to interface other Windows applications with Arbortext Editor.

Note

Consult your Windows developer documentation for details on using the DDE layer for inter-application communications.

The first instance of Arbortext Editor acts as a DDE server, and will accept connections which specify a service of `adept`. Arbortext Editor's DDE services support the following topics:

- `system` — When you connect to Arbortext Editor using the `system` topic, Arbortext Editor will accept `execute` transactions. If the message text is `/exec`, Arbortext Editor does nothing. This is the command used by the `.sgm` file association to start Arbortext Editor when you double-click an `.sgm` file. If the message text is `/activate`, Arbortext Editor will bring itself to the foreground.
- `eval` — When you connect to Arbortext Editor using the `eval` topic, Arbortext Editor will accept `request` transactions. The message text will be evaluated as an ACL function expression; any string resulting from the expression is returned to the sender application. This functionality is similar to executing an `eval` command at the Arbortext Editor command line, with the returned value being sent to the requesting application instead of being displayed in a dialog.

  While the messages sent with the `eval` topic only support ACL functions, you can send a command by calling it with the `execute` on page 351 ACL function.

Built-in Functions, Callbacks, and Hooks

Built-in functions, callbacks, and hooks are built into Arbortext Editor and may be used in expressions.

Use either of the following methods to display the help for a specific command:

- At the **Command** line, enter the command `help` *`commandname`*, where *commandname* is the name of the desired command, function, callback, or hook.
- In the Help Center, search for the name of the command, function, callback, or hook to view any topics mentioning the term.

Array, Variable, and Package Functions

Array, variable, and package functions operate on ACL features (arrays, variables, functions, and packages) or provide information about commands and options.

Buffer Functions

Buffer functions operate on named paste buffers.

Byte String Functions

Byte string functions operate on byte strings. Byte strings are used to represent binary data as opposed to character strings which represent Unicode strings (two bytes per character).

Context-Related Functions

Context-related functions pertain to the context for a given element in a DTD or at a point within a specific document.

Dialog Box Functions

Dialog box functions provide dialog boxes for user interaction.

Document Identifier Functions

Most of the Document type functions also take an optional final argument, which is a document ID. Document Identifier functions return or take document identifiers.

Document Type Functions

Document type functions return information about a particular document type. Most functions take an optional final argument. This final argument is a document identifier that determines the document type.

Dynamic Loading (API) Functions

Dynamic loading functions provide a direct interface to the operating system facilities for loading, calling, and unloading functions in dynamically loaded (shared) libraries. In the Windows environment, this includes the following

dynamic link library APIs: `LoadLibrary`, `FreeLibrary` and `GetProcAddress`. These functions also allow you to extend ACL by writing DLLs in C or in other programming languages that support DLLs.

Editor Functions

Editor functions query or change the state of a document instance. The OID functions that change the document tree are also found in the “Object identifier (OID) functions” topic.

Entity Functions

Entity functions provide information about general text entities or external file entities.

File Identifier Functions

File identifier functions manipulate file identifiers (FIDs) returned by the `open` function. These functions provide an interface to the C-library standard I/O routines.

File or System Functions

File or system functions provide file or system information or are used to manipulate pathnames.

Menu and Keymap Functions

The menu and keymap functions query keymaps or the menu system.

Notation Functions

Notation functions provide information about notations declared in a document.

Table Functions

Table functions edit tables in the Edit window.

Table Object and Table Model IDs

Functions that take or return *table object identifier* (*toid*) or *table model identifier* (*tmid*) parameters or values may use these predefined constants to test for null or invalid values:

- `h::tblNullToid`
- `h::tblNullTmid`

Table Object Types

Functions that take or return *objectType* parameters or results may use these predefined values:

- `h::tblTypeUndef`
- `h::tblTypeSet`
- `h::tblTypeGrid`
- `h::tblTypeRow`
- `h::tblTypeColumn`
- `h::tblTypeCell`
- `h::tblTypeRule`
- `h::tblTypeStore`
- `h::tblTypeSelector`

Table Directions

Functions that take or return directions may use these predefined values:

- `h::tblDirUndef`
- `h::tblDirLeft`
- `h::tblDirRight`
- `h::tblDirAbove`
- `h::tblDirBelow`

Table Orientation

Functions that take or return orientations may use these predefined values:

- `h::tblVertical`
- `h::tblHorizontal`

List Functions

Functions whose names end with “list” take as one parameter an array variable. On return they place some number of data elements in the array and return the number of array elements as their result. The list will be numerically indexed. If the function is unable to return any information, it will return zero, indicating that there is nothing in the array.

Row-Major Ordering

Functions that return a list of cells or rules usually do so in row-major order. Row major order means starting with the upper-left cell in a grid (or in a region within a grid) and proceeding horizontally from left to right. At the end of the row, we move to the left most cell of the row below, and again continue from left to right. The last cell processed is the lower-right cell of the grid or region.

Galley Ordering

Some routines navigate around a table or grid using “galley” order. Galley order is row-major order with one difference: cells that are spanned and which are not the master cell are omitted entirely. The basic progression (start with upper-left cell, continue from left to right in each row, process rows from top to bottom, ending with the lower-right cell) is the same, we just skip cells whose content is uninteresting because they're neither unspanned nor the master cell of a multicell.

Error Handling

Functions that return *toid* or *tmid* values will return `h::tblNullToid` or `h::tblNullTmid` to indicate failure.

Functions that return [`0`|`1`] will return `0` to indicate error.

List functions return `0` to indicate that no data was returned in an array.

Table Object Attributes

Attribute values are initially derived from table markup. Values may be manipulated by the following ACL functions:

- tbl_obj_attr_clear on page 562
- tbl_obj_attr_delete on page 562
- tbl_obj_attr_get on page 562
- tbl_obj_attr_set on page 563

Some values are numbers, some values are strings. String values are case insensitive.

Mandatory Attribute Values

Table object attributes always have a value, even if the value is meaningless. ACL programmers should account for this. For example, if the table markup specifies no row height, Arbortext Editor wants to display the row with the minimum height needed to handle the contents of the cells in the row. Since there's always a value for a row's `TARGET_HEIGHT` we provide a second attribute, `USE_TARGET_HEIGHT`, which is zero (`0`) or one (`1`) to indicate whether the value in `TARGET_HEIGHT` should be used or ignored.

Dimensions in Attributes

A number of attributes specify dimensions: 3 inches, 2 pixels, etc. Such attribute values are specified as a “unitdim”, a floating-point number followed by a unit (for example, `3.0in`, `2.0px`). Valid units are `in`, `pt`, `pi`, `px`, `cm`, `mm`, or `%`, meaning inches, points, picas, pixels, centimeters, millimeters, or percent.

Shared Table Attributes

The following attributes are supported on all table objects (sets, grids, columns, rows, cells, and rules).

OID

This attribute identifies the markup tag corresponding to the set, grid, row, column, cell or rule. Depending upon the table model, this value may be null. For example, there's no tag corresponding to a column under the Arbortext table model.

The user is not allowed to set this attribute.

RESIZABLE

This attribute is zero (`0`) for not resizable or one (`1`) for resizable. It is always zero (`0`) for a set, cell or rule, as these items are not considered resizable. For a grid, row, or column, this value indicates whether the width (grid or column) or height (row) may be changed. The value will be zero (`0`) if the relevant parts of the document are read-only or if a variety of other internal conditions apply.

The user is not allowed to set this attribute.

Set Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attributes are supported on set table objects.

CELLPADDING

This attribute applies to HTML tables only.

This attribute controls the space between cell wall and the cell contents (the cell wall is at the center of the cell border, not at its edge). The only supported unit is `px` (pixel). Enter a whole number value in the field provided. The default value is `2` pixels.

CELLSPACING

This attribute applies to HTML tables only.

This attribute controls the space between cells and between the table outer border and the table cells. The only supported unit is `px` (pixel). Enter a whole number value in the field provided. The default value is `1` pixel.

DEFAULT_RTH_UNIT

This attribute indicates the default "row target height" unit. If a row has an explicit height, for example `1.0in`, and the user drag-resizes the row's height, we store the new height to markup using `in` (inches) as the unit.

Consider the case, however, in which a table has no explicit row heights given in its markup. If the user drag-resizes a row, Arbortext Editor wants to store the value thus created, but doesn't know which unit to use. This attribute provides an answer to the question.

If a row has no explicit height set and a user's edit forces Arbortext Editor to pick a unit, it uses the unit named by this attribute on the Set containing the Grid containing the Row.

Permissible units in OASIS Exchange and Arbortext tables are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, the only allowable row height unit is pixels. Consequently, this attribute is not supported in HTML tables.

FRAME

This attribute indicates how the outer borders of a Set are drawn. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Value	Effect
`top`	Draws the top border of the set only.
`bottom`	Draws the bottom border of the set only.
`topbot`	Draws the top and bottom borders of the set only.
`all`	Draws all four borders of the set.

Value	Effect
`sides`	Draws the left and right borders of the set only.
`left`	Draws the left border of the set only.*
`right`	Draws the right border of the set only.*
`none`	Draws no borders.

*Applies to HTML tables only.

FRAME_COLOR_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the top, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the bottom, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the left, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the right, outer border. The value is either a named color or an RGB value.

FRAME_STYLE_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the top, outer border. The following values are supported:

- `blank`
- `single`
- `double`
- `dashed`
- `dotted`

FRAME_STYLE_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the bottom, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_STYLE_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the left, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_STYLE_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the right, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_WIDTH_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the top, outer border. The value is a decimal number followed by a unit of measure, for example `1.0pt`. The following units of measurement are supported:

- `pt` — points
- `pi` — picas
- `in` — inches
- `mm` — millimeters
- `cm` — centimeters
- `px` — pixels.
- `em` — em space

FRAME_WIDTH_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the bottom, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAME_WIDTH_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the left, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAME_WIDTH_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the right, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAMERULES

This attribute applies to custom tables only.

This attribute sets the rules that are drawn for the custom table. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Value	Effect
`all`	Draws all rules. (the default)
`top`	Draws the top frame rule only.
`bottom`	Draws the bottom frame rule only.
`topbot`	Draws the top and bottom frame rules only.
`left`	Draws the left frame rules only.
`right`	Draws the right frame rules only.
`sides`	Draws the left and right frame rules only.
`rows`	Draws the horizontal rules only.
`cols`	Draws the vertical rules only.
`frame-only`	Draws the outer frame rules only.
`none`	Draws no rules.

FRAMETHICKNESS

This attribute applies to HTML tables only.

This attribute sets the width of the outer table border. The only supported unit is `px` (pixels). Attribute values must be whole numbers.

GUITYPE

This attribute returns the string `oasis`, `ati`, or `html` to specify that the table editor is to use the Oasis Exchange, Arbortext, or HTML (respectively) dialog box options for the table.

The user is not allowed to set this attribute.

RULE

This attribute applies to HTML tables only.

This attribute sets which rules are drawn for the inside of the table. The outer borders are controlled by the `FRAME` attribute. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Supported values for `RULE` attribute

`all`	Draws all the internal rules within the set.
`rows`	Draws only the horizontal rules between the rows within the set.

Supported values for `RULE` attribute (continued)

`cols`	Draws only the vertical rules between the columns within the set.
`groups`	Draws a single horizontal rule between the header and body rows and another single horizontal rule between the body and footer rows.
`none`	Draws no rules within the set.

TMID

This attribute is the Table Model ID of the Table Model that is managing the Set and its content.

The user is not allowed to set this attribute.

Grid Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attributes are supported on grid table objects.

RULETHICKNESS

This attribute is a unitdim value indicating the thickness of the rules. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

USEWIDTH

This attribute is zero (`0`) or one (`1`), indicating whether the Grid's `WIDTH` attribute should be ignored (`0`) or used (`1`). While the user cannot set this attribute directly, if they explicitly set the `WIDTH` attribute, this attribute's value automatically changes to one (`1`). If the user resets the `WIDTH` attribute to the default value, this attribute value automatically changes to zero (`0`).

WIDTH

If the `USEWIDTH` attribute is a one (`1`), this attribute is a unitdim indicating the width of the Grid on the screen. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

Note

This attribute (or, more precisely, the markup underlying it) is not used in published output.

Column Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attributes are supported on column table objects.

WIDTH_MAX

This attribute is a unitdim indicating the maximum width to which the column can be drag-resized. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

WIDTH_MIN

This attribute is a unitdim indicating the minimum width to which the column can be drag-resized. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

COLWIDTH

This attribute is a unitdim indicating the column's width. In addition to the units allowed for other unitdims (that is, `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters), the column's width unit may also be `*`, indicating proportional width units.

WIDTH_UNIT

This attribute is the unit portion of the `COLWIDTH` attribute. This attribute exists to allow the user to convert between units without doing actual calculations. If a column's `COLWIDTH` attribute is `1.0in`, a call to the `tbl_obj_attr_set` function with a `WIDTH_UNIT` attribute value of `cm` will cause the column's `COLWIDTH` unit to be changed to `2.54cm`.

Row Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attributes are supported on row table objects.

BREAK_PENALTY

The value of `BREAK_PENALTY` is used to determine whether to cause a page break. `BREAK_PENALTY` can be one of the following values:

- `default` — No preference on a break.
- `allow` — Permit a break.
- `inhibit` — Do not permit a break.
- `force` — Insert a break.
- `recto` — Break to the next odd page.
- `verso` — Break to the next even page.

FOOTER_LEVEL

If the row is a footer row, this attribute will be one (`1`) otherwise it will be zero (`0`). This attribute cannot be set to one (`1`) if the `HEADER_LEVEL` attribute is already set to one (`1`).

For the Arbortext table model, no row may have a non-zero footer level. For the CALS table model (OASIS Exchange), the table can include a block of header rows, followed by a block of footer rows, followed by a block of body rows. For the CALS table model, setting a row's footer level will cause the row to be moved to the footer or body block, as appropriate.

You can not mark a row as a footer row if doing so removes the only row from the table that is not a footer row. For example, if there is only one row in the table, it can't be a footer row as there must always be at least one row in the table that is not a footer row.

Do not change the value of `FOOTER_LEVEL`. Doing so will render the original row invalid and may cause an AOM exception to be thrown or cause Arbortext Editor to terminate unexpectedly.

HEADER_LEVEL

If the row is a header row, this attribute will be one (`1`), otherwise it will be zero (`0`). This attribute cannot be set to one (`1`) if the `FOOTER_LEVEL` attribute is already set to one (`1`).

For the Arbortext table model, only the first row in the table may have a nonzero header level. For the CALS table model (OASIS Exchange), the table can include of a block of header rows, followed by a block of footer rows, followed by a block of body rows. For the CALS table model, setting a row's header level will cause the row to be moved to the header or body block, as appropriate.

You can't mark a row as a header row if this action removes the only row from the table that is not a header row. For example, if there is only one row in the table, it can't be a header row as there must always be at least one row in the table that is not a header row.

Do not change the value of `HEADER_LEVEL`. Doing so will render the original row invalid and may cause an AOM exception to be thrown or cause Arbortext Editor to terminate unexpectedly.

HEIGHT_UNIT

This attribute is the unit portion of the `TARGET_HEIGHT` attribute. This attribute exists to allow the user to convert between units without doing actual calculations. If a row's `TARGET_HEIGHT` attribute is `1.0in`, a call to the `tbl_obj_attr_set` function with a `HEIGHT_UNIT` attribute value of `cm` will cause the row's `TARGET_HEIGHT`unit to be changed to `2.54cm`.

TARGET_HEIGHT

If the row's `USE_TARGET_HEIGHT` attribute is one (`1`), this attribute is a unitdim value indicating the height that is used to display the row. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

Setting this attribute causes the `USE_TARGET_HEIGHT` attribute to be set to one (`1`). Resetting this attribute to its default value (using `tbl_obj_attr_delete`) causes the `USE_TARGET_HEIGHT` attribute to be set to zero (`0`).

USE_TARGET_HEIGHT

This attribute may be zero (`0`) or one (`1`) to indicate whether the value of the `TARGET_HEIGHT` attribute should be used. The user can not set this attribute directly, but can influence its value by calling `tbl_obj_attr_get` or `tbl_obj_attr_delete` for the `TARGET_HEIGHT` attribute.

Cell Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attributes are supported on cell table objects.

BGCOLOR

This attribute controls the background color of the cell. The value is either a named color or an RGB value.

BOTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's bottom rule and the edge of the box in which text is actually displayed. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (`px` is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the `CELLPADDING` set table attribute.

LEFTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's left rule and the edge of the box in which text is actually displayed. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (`px` is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the `CELLPADDING` set table attribute.

RIGHTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's right rule and the edge of the box in which text is actually

displayed. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (`px` is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the `CELLPADDING` set table attribute.

TOPMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's top rule and the edge of the box in which text is actually displayed. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (`px` is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the `CELLPADDING` set table attribute.

HALIGN

This attribute controls horizontal text justification for the cell. Permissible values are `left`, `center`, `right`, `justified`, or `character`.

The following values are also valid, but cannot be used to control the horizontal text justification for the cell: `start`, `end`, or `inherit`.

Character alignment is not effective in HTML outputs. Browsers are not required to support it, and they do not.

HALIGN_CHAR

This attribute is a single character on which to horizontally align the text in a column of cells. This character alignment scheme is used only if the `HALIGN` attribute is set to `character`.

HALIGN_CHAR_OFF

This attribute is a number between zero (`0`) and `100` indicating the desired location of the alignment character given by the `HALIGN_CHAR` attribute. Zero (`0`) places the character to the extreme left side of the cell, `100` places the character to the extreme right of the cell. This offset is used only if the `HALIGN` attribute is set to `character`.

HEADER

This attribute applies only to HTML tables. This attribute is 0 if the cell is a data cell (td) and is 1 if the cell is a header cell (th).

RULECOLOR_ABOVE

This is a convenience attribute to allow the `COLOR` attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_BELOW

This is a convenience attribute to allow the `COLOR` attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_LEFT

This is a convenience attribute to allow the `COLOR` attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_RIGHT

This is a convenience attribute to allow the `COLOR` attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_ABOVE

This is a convenience attribute to allow the `STYLE` attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_BELOW

This is a convenience attribute to allow the `STYLE` attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_LEFT

This is a convenience attribute to allow the `STYLE` attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_RIGHT

This is a convenience attribute to allow the `STYLE` attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_ABOVE

This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_BELOW

This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_LEFT
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_RIGHT
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

VALIGN
This attribute is either `top`, `center`, or `bottom` to indicate vertical alignment of the contents of the cell.

Rule Table Attributes

In addition to the attributes listed in the Shared table attributes on page 124 help topic, the following attribute is supported on rule table objects.

COLOR
This attribute indicates the color of the rule should be drawn. The value is either a named color or an RGB value.

RULEWIDTH
This attribute indicates the width of the rule. The value is a decimal number followed by a unit of measure, for example `1.0pt`. The following units of measurement are supported:

- `pt` — points
- `pi` — picas
- `in` — inches
- `mm` — millimeters
- `cm` — centimeters
- `px` — pixels.
- `em` — em space

STYLE
This attribute indicates how the rule should be drawn. The following values are supported:

- `blank`
- `single`
- `double`

- `dashed`
- `dotted`

Time Functions

Time functions return time information or create timers.

Arbortext Editor Command Aliases

For help on a built-in command alias, double-click on an alias name:

Set Option Scope

Each set option has a scope which defines how it can affect your Arbortext Editor session. Set options use the following scope definitions:

- Session — Almost all set options have a session scope. The session scope encompasses all the views in all the windows of the current Arbortext Editor session. The `catalogpath` and `graphicspath` set options are examples

of set options that have a session scope, but no local scope; such options are said to be “global” options.

- Preference — Not all set options are saved in the preference file. If a set option is a preference, then it is used to create the first edit window when Arbortext Editor is started. Changing the local or session scope of a set option does not change its preference setting. Use the `write_preferences` function to set preferences.
- Local — Some set options have a local scope, meaning that the set option is controllable locally, without changing the session setting. The local scopes are:
    - Window — If a set option is applicable to all views (split windows) in a Arbortext Editor window, it has a window local scope. The `toolbar1` set option is an example of a set option with a local scope of window. Changing the local scope of `toolbar1` to `off` would turn off the edit toolbar for the currently active window, leaving it on in the other open windows. Changing the session scope of `toolbar1` to `off` would turn off the edit toolbar for all currently open windows and any new windows.
    - Document — If a set option is applicable to all windows/views for a given document, it has a document local scope. The `entitylist` set option is an example of a set option with a local scope of document. Changing the local scope of `entitylist` to include the `alpha` character entity would add the `alpha` character entity to the pulldown list for all windows that contained the current document, but not affecting the pulldown lists for other open documents or subsequently opened documents. Changing the session scope of `entitylist` to include the `alpha` character entity would add the `alpha` character entity to the pulldown list for all open documents and subsequently opened documents.
    - View — If a set option is applicable to only the current view in a Arbortext Editor window, it has a view local scope. The `tagdisplay` set option is an example of a set option with a local scope of view. Changing the local scope of `tagdisplay` to `full` would display the full markup tags for the currently active view, no tags visible in the other views (in a split window scenario), and in other windows. Changing the session scope of `tagdisplay` to `full` would display full markup tags for all the views in all the currently open windows and any new windows.

show commands

Use the `show` command to display information about an *option*. The syntax for the show command is `show` *option*.

Following is a list of supported options for the `show` command:

Creating and Manipulating Windows

An unlimited number of application-defined windows may be created using the function `window_create`. The class (or type) of the window is specified as the first argument to `window_create`. It may be one of the strings: “edit,” “msg,” “msgwin1 ” through “msgwin4,” “help,” “helpwin1 ” through “helpwin4,” or “list.”

The class determines the default geometry and whether a document tree is associated with the window. If the class is list , then a list selection dialog is created. Otherwise, the window displays a document tree and has a class-specific keymap associated with it. For document class windows (all except list class), the second argument to `window_create` is a bit mask of flags that controls whether a menu bar is created for the window, whether a vertical scrollbar is created, whether a message area is provided at the bottom of the window, and for edit class windows, whether a Command window is supplied below the Edit subwindow.

The third argument to `window_create` is a document identifier obtained from a previous call to `doc_open` or if -1, then a scratch document (doctype ascii for edit classes, or the built-in help document type for other classes) is created and attached to the window. (The associated document tree may be changed later by calling `doc_show`.) Since `doc_open` supports arbitrary document types, windows created using `window_create` may display different arbitrary document types, and in fact, for edit classes, full interactive context checking is enabled. There is a slight performance slowdown when switching between document types, since the parser does not support simultaneous parsing of disparate document types. Internally, the parser is reinitialized when switching between document types, which typically takes one second or so.

The window names edit and cmd do not specify unique windows. These are window classes and when used on commands like `caret` refer to a window relative to the current document's window, or on the `map` command refer to a

window class map. For example, `caret edit` issued from a command window keymapping, moves the focus to the corresponding Edit window. If issued from outside an edit or cmd -class window, it will give focus to the last edit class window having focus. The window names msg, msgwin [1-4], help, and helpwin [1-4] refer to the built-in fixed windows (for backward compatibility). These names, when used on commands like `caret`, or as a *window* argument to the various `window_`*xxx* functions like `window_close`, always refer to the built-in windows and never to window of that class created by `window_create`. A window ID must be used to refer to such windows created by `window_create`.

There are several window manipulation functions, which are listed in the Window manipulation functions on page 139 table.

- Example of edit window function on page 139
- Example of list response panel on page 140
- `window_class` built-in function on page 587
- `window_get` built-in function on page 592
- `window_id` built-in function on page 593
- `window_name` built-in function on page 596

Window Manipulation Functions

Refer the topics in the *Window functions* section of the online help, beginning with the topic current_event on page 255, for a listing of the available window manipulation functions. Also refer to the following examples:

- Example of edit window function on page 139
- Example of list response panel on page 140

Example: Edit Window Function

Here is an example of a function that could be used to edit another document in a new window. This function is defined in the system `_main.acl` file. (Note: flags equals the sum of bits set.)

```
function edit_new_window(file, flags=0x3f, geom="") {
# if no special args, use builtin command
if (flags&0x3f == 0x3f && geom == "") {
 edit -newwindow $file
 return}
# open a new doc tree
local doc = doc_open(file);
if (doc < 0) {
 # error message already displayed by doc_open
 return}
# see if already in window; if so just expose it
```

```
local win = doc_window(doc);
if (win > 0) {
 doc_close(doc);
 # decrement reference count
 window_open(win);
 # uniconify maybe
 window_raise(win);
 return}
# make an edit class window for it with
# edit-style initialization flags |= 0x140
win = window_create("edit", flags, doc, geom);
 if (win < 0) { message "Couldn't create
 window to edit: $file";
 # couldn't make a window so unload document
 doc_close(doc);
 return}
# make this doc be the current one for
# the set command
current_doc(doc)
# convert tagged tables to pictures according
# to global setting
if (option("tabletags") == "off") {
 set tabletags=off}
 window_show(win, 1);
# map the window now}
```

Example: List Response Panel

You can create any number of non-blocking list response panels using `window_create`. The `window_set` function is used to change various attributes of the panel, for example:

- the list of items to display
- the label at the top of the panel
- the default input item
- the function to call when a button is pressed
- the labels displayed on the buttons

The scrolling list may be set from an array or appended one item at a time. The callback function has complete control over the behavior of the window when a button is pressed. For example, it can choose to unmap the window (for reuse), destroy it, or display a message and leave the window up.

Here is an example.

```
function list_callback(win, but, seln, pos)
{
if (but == 3)
 {
```

```
 # HELP
 window_set(win, "message", "Pick any number");
}
else if (but == 2)
 {
 # APPLY: leave panel up
 # ... do something here with $seln
}
else if (but == 1)
 {
 # OK
 # take down window, but don't destroy it for reuse
 window_show(win, 0);
 # ... do something here with $seln}
else
 {
 # else 0 (CANCEL) or -1 (window manager quit)
 window_destroy(win);
}
return 0;
}
function pickanum()
{
 local w, X[]
 # make an array of items
 split("one two three four", X)
 w = window_create("list")
 window_set(w, "items", X, "default", X[2], \
 "title", "Pick a Number", \
 "callback", "list_callback")
 window_show(w, 1);
 return w;
}
```

3

Functions by Category

Alias Map Functions

An alias map maps elements, attributes, and attribute values to new names or values. The alias map functions return the alias of the specified function as defined in the DTD.

- `attr_alias` function on page 227
- `attr_description` function on page 227
- `attr_real_name` function on page 228
- `attr_value_alias` function on page 228
- `attr_value_real_name` function on page 229
- `tag_alias` function on page 531
- `tag_description` function on page 538
- `tag_real_name` function on page 542

Applicability Functions

- `registerApplicabilitySyntax` function on page 504

Application Management Functions

- `application_name` function on page 224
- `doc_clone` function on page 314
- `doc_dir` function on page 316
- `get_custom_dir` function on page 366
- `get_custom_property` function on page 367
- `get_user_property` function on page 369
- `set_user_property` function on page 515

Array, Variable, and Package Functions

Array, variable, and package functions operate on ACL features (arrays, variables, functions, and packages) or provide information about commands and options.

- `append_catalog_path` function on page 218
- `append_composer_path` function on page 219
- `append_dialogs_path` function on page 219
- `append_dita_path` function on page 220

- append_entity_path function on page 220
- append_frameset_path function on page 221
- append_graphics_path function on page 221
- append_javaclass_path function on page 221
- append_load_path function on page 222
- append_newlist_path function on page 223
- append_path function on page 224
- append_tagtemplate_path function on page 224
- append_userdict_path function on page 223
- buffer_empty function on page 231
- caller function on page 234
- caller_file function on page 234
- caller_line function on page 234
- catch function on page 238
- cmd_exists function on page 244
- cmd_key function on page 244
- count function on page 253
- defined function on page 263
- delete function on page 264
- eval (Function) function on page 349
- execute (Function) function on page 351
- graphic_views function on page 378
- high_bound function on page 380
- low_bound function on page 407
- option function on page 474
- package_file function on page 480
- package_name function on page 480
- package (Function) function on page 480
- packages function on page 481
- qsort function on page 501
- require (Function) function on page 506
- throw function on page 575
- varsub function on page 585

Buffer Functions

Buffer functions operate on named paste buffers.

- `buffer_clipboard_contents` function on page 230
- `buffer_clipboard_formats` function on page 231
- `buffer_create` function on page 231
- `buffer_doc` function on page 231
- `buffer_empty` function on page 231
- `buffer_is_clipboard` function on page 232
- `buffer_is_table` function on page 232
- `insert_buffer` function on page 385

Byte String Functions

Byte string functions operate on byte strings. Byte strings are used to represent binary data as opposed to character strings which represent Unicode strings (two bytes per character).

- `blength` function on page 229
- `chop` function on page 242
- `index` function on page 384
- `length` function on page 399
- `mblen` function on page 412
- `mbstoucs` function on page 412
- `oct` function on page 427
- `pack` function on page 478
- `rindex` function on page 507
- `substr` function on page 531
- `trim` function on page 577
- `ucstombs` function on page 577
- `unpack` function on page 579

Callback Functions

A callback function is a user-defined function that is called during specific events in Arbortext Editor and allows you to customize editing operations. Callback functions provide both a global and a local level of registration or notification to

isolate events, while hooks provide only global effect (unless additional user functions are supplied). For example, you might want a particular function called whenever a specific document is saved or a help window has a quit request.

- `channel_set_callback` function on page 976
- `dlgitem_add_callback` function on page 978
- `dlgitem_remove_callback` function on page 979
- `doc_add_callback` function on page 980
- `doc_remove_callback` function on page 1022
- `session_add_callback` function on page 1023
- `session_remove_callback` function on page 1026
- `timer_add_callback` function on page 1026
- `timer_remove_callback` function on page 1027
- `userule_add_callback` function on page 1027
- `userule_remove_callback` function on page 1027
- `window_add_callback` function on page 1028
- `window_remove_callback` function on page 1030

Change Tracking Functions

The change tracking feature in Arbortext Editor tracks text and markup changes made to a baseline document. These functions attempt to accept, reject, locate and collect information from the baseline document.

- `change_tracking_accept_change` function on page 238
- `change_tracking_accept_selection` function on page 238
- `change_tracking_find` function on page 239
- `change_tracking_info` function on page 240
- `change_tracking_reject_change` function on page 240
- `change_tracking_reject_selection` function on page 241
- `change_tracking_user_list` function on page 241
- `change_tracking_user_properties` function on page 242
- `doc_has_change_tracking` function on page 321
- `selection_has_change_tracking` function on page 512

Channel Functions

The channel functions provide interfaces to the Berkeley socket library, which allows network applications to be written. The `open` functions return channel identifiers that may also be used with most file identifier functions (for example, `read`, `write`, and `close`). Several `open_` routines return a channel identifier that may be used by subsequent I/O functions. A channel identifier is a small integer file identifier that is returned by `open` and can be used with the file identifier routines `getline`, `put`, `close`, `read`, and `write`. These channel functions allow binary data to be transmitted.

- `open_accept` function on page 471
- `open_connect` function on page 471
- `open_listen` function on page 472
- `pack` function on page 478
- `read (Function)` function on page 502
- `unpack` function on page 579
- `write (Function)` function on page 607

Column View Functions

Column view is a Arbortext Editor view specifically designed to support document types that primarily consist of element and attribute content. These functions enable you to operate in column view.

- `columnview_cell_focus` function on page 245
- `columnview_is_defined` function on page 245
- `window_get_columnview` function on page 593
- `window_set_columnview` function on page 603

COM Interface Functions

These functions support COM (Component Object Model) calls.

- `com_attach` function on page 246
- `com_call` function on page 246
- `com_prop_get` function on page 247
- `com_prop_put` function on page 248
- `com_release` function on page 248

Composer Functions

These functions support operations in a log file.

- composer_log function on page 249
- get_composer_log_contents function on page 365
- get_composer_log_doc function on page 366
- show_composer_log function on page 516

Context-Related Functions

Context-related functions pertain to the context for a given element in a DTD or at a point within a specific document.

- content_model function on page 250
- context_full_paths function on page 251
- context_paths function on page 252
- context_string function on page 253
- cut_valid function on page 257
- delete_markup_valid function on page 265
- in_context function on page 383
- in_context_list function on page 384
- oid_expose function on page 439
- oid_paste_valid function on page 454
- oid_show_attr function on page 459
- tag_content function on page 537

Custom Dialog Box Functions

The Arbortext XUI technology lets an application programmer create, display, manipulate, and modify dialog boxes in real time by writing and modifying XML documents. Refer to the *Customizer's Guide* for details on working with XUI dialog boxes.

Windows Functions

Windows functions operate on the attribute of the window to be changed.

- window_load_component_file on page 595

Dialog Item Functions

Dialog item functions operate on dialog items (controls). Each dialog item is identified by a window and an ID. IDs must be unique and case sensitive within the dialog box.

- dlgitem_dropdown_list function on page 279
- dlgitem_ensure_listtag_visible function on page 280
- dlgitem_ensure_table_visible_at function on page 280
- dlgitem_find_table_cell_in_column function on page 281
- dlgitem_get function on page 281
- dlgitem_get_all function on page 282
- dlgitem_get_background_at function on page 283
- dlgitem_get_focus function on page 284
- dlgitem_get_foreground_at function on page 284
- dlgitem_get_list_array function on page 285
- dlgitem_get_list_at function on page 285
- dlgitem_get_mnemonic function on page 286
- dlgitem_get_select_array function on page 287
- dlgitem_get_table_cell_at function on page 289
- dlgitem_get_table_column_align function on page 289
- dlgitem_get_table_column_count function on page 290
- dlgitem_get_table_column_header_at function on page 290
- dlgitem_get_table_row_count function on page 290
- dlgitem_get_table_selection function on page 291
- dlgitem_get_table_sort function on page 291
- dlgitem_get_value function on page 291
- dlgitem_insert_table_column_at function on page 293
- dlgitem_insert_table_row_at function on page 293
- dlgitem_is_active function on page 293
- dlgitem_remove_table_column_at function on page 295
- dlgitem_remove_table_row_at function on page 295
- dlgitem_remove_toolbar function on page 295
- dlgitem_set function on page 296

- dlgitem_set_background_at function on page 300
- dlgitem_set_focus function on page 302
- dlgitem_set_foreground_at function on page 303
- dlgitem_set_list_array function on page 303
- dlgitem_set_list_at function on page 304
- dlgitem_set_mnemonic function on page 307
- dlgitem_set_multiple function on page 307
- dlgitem_set_table_cell_at function on page 308
- dlgitem_set_table_column_align function on page 309
- dlgitem_set_table_column_count function on page 309
- dlgitem_set_table_column_header_at function on page 309
- dlgitem_set_table_row_count function on page 310
- dlgitem_set_table_selection function on page 310
- dlgitem_set_table_sort function on page 310
- dlgitem_set_value function on page 311

Convenience Functions

The convenience functions are defined in terms of other functions. They are provided as an simple and obvious alias for other, more general functions.

- dlgitem_activate function on page 277
- dlgitem_deactivate function on page 278
- dlgitem_display function on page 278
- dlgitem_get_appdata function on page 282
- dlgitem_get_list_count function on page 285
- dlgitem_hide function on page 292
- dlgitem_set_appdata function on page 299
- dlgitem_set_list_count function on page 304
- dlgitem_show function on page 312
- dlgitem_withdraw function on page 312

Outline List Item Functions

The outline list item functions operate on outline lists, and will fail if applied to other types of dialog items. Note that the EXPAND attribute is specific to outline lists, and denotes the function to be called when a node must be expanded.

- dlgitem_collapse function on page 277
- dlgitem_exists function on page 280
- dlgitem_expand function on page 280
- dlgitem_get_active_at function on page 281
- dlgitem_get_appdata_at function on page 283
- dlgitem_get_check_at function on page 283
- dlgitem_get_listtag function on page 286
- dlgitem_get_listtag_by_appdata function on page 286
- dlgitem_get_selected_appdata function on page 288
- dlgitem_get_selected_listtag function on page 288
- dlgitem_get_selected_listtag_array function on page 288
- dlgitem_get_sublisttag function on page 289
- dlgitem_insert_list_at function on page 292
- dlgitem_is_expandable function on page 294
- dlgitem_is_expanded function on page 294
- dlgitem_remove_list_at function on page 294
- dlgitem_select_list_at function on page 296
- dlgitem_set_active_at function on page 299
- dlgitem_set_appdata_at function on page 299
- dlgitem_set_check_at function on page 300
- dlgitem_set_default_branch_image function on page 301
- dlgitem_set_default_leaf_image function on page 301
- dlgitem_set_default_openbranch_image function on page 302
- dlgitem_set_listtag_branch_image_at function on page 305
- dlgitem_set_listtag_extra_image_at function on page 305
- dlgitem_set_listtag_leaf_image_at function on page 305
- dlgitem_set_listtag_openbranch_image_at function on page 306
- dlgitem_set_refresh function on page 308
- dlgitem_set_selection function on page 308

Dialog Box Functions

Dialog box functions provide dialog boxes for user interaction.

- color_chooser function on page 244
- file_selector function on page 355
- list_response function on page 403
- panel_popup function on page 481
- response function on page 506

Document Identifier Functions

Most of the Document type functions also take an optional final argument, which is a document ID. Document Identifier functions return or take document identifiers.

- current_doc function on page 255
- doc_cache_base function on page 313
- doc_cache_dir function on page 313
- doc_clean_cache function on page 313
- doc_clone function on page 314
- doc_close function on page 315
- doc_delete_stylesheet_association function on page 316
- doc_estimate_dfs function on page 317
- doc_formattable function on page 319
- doc_get_stylesheet_association function on page 320
- doc_incomplete function on page 321
- doc_invalidate_graphics function on page 322
- doc_kind function on page 323
- doc_list function on page 323
- doc_name function on page 324
- doc_new_stylesheet_association function on page 324
- doc_num_stylesheet_associations function on page 325
- doc_open function on page 325
- doc_parent function on page 328
- doc_path function on page 328
- doc_read_only function on page 328
- doc_revert function on page 329
- doc_save function on page 329

- doc_set function on page 330
- doc_set_path function on page 331
- doc_set_stylesheet_association function on page 331
- doc_set_translation_locale function on page 332
- doc_set_translation_orig_uri function on page 332
- doc_show function on page 332
- doc_update_display function on page 336
- doc_valid function on page 336
- doc_window function on page 336
- path_doc function on page 481
- save_as_html_file function on page 507
- save_some_docs function on page 508

Document Type Functions

Document type functions return information about a particular document type. Most functions take an optional final argument. This final argument is a document identifier that determines the document type.

- base_tag_name function on page 229
- bulleted_list_block_tag_name function on page 232
- bulleted_list_block_tag_name_ns function on page 233
- bulleted_list_item_tag_name function on page 233
- bulleted_list_item_tag_name_ns function on page 233
- catalog_reslove function on page 288
- char_entity_names function on page 242
- content_model function on page 250
- dcf_option function on page 257
- dcf_validate function on page 257
- dcfmodel_element_list function on page 260
- ditaref_resolve function on page 269
- division_heading_tag function on page 270
- division_tag function on page 271
- doc_compose_stylesheets function on page 315
- doc_dtd_not_defined function on page 317

- doc_dtd_not_found function on page 317
- doc_freeform function on page 319
- doc_namecase_sensitive function on page 324
- doc_type function on page 333
- doc_type_dcf_file function on page 333
- doc_type_description function on page 333
- doc_type_dir function on page 333
- doc_type_dita function on page 333
- doc_type_extension function on page 334
- doc_type_file function on page 334
- doc_type_gi function on page 334
- doc_type_language function on page 334
- doc_type_namespace function on page 334
- doc_type_namespaces function on page 335
- doc_type_xml function on page 336
- dtd_decl_path function on page 341
- dtd_tag function on page 341
- entity_tag function on page 348
- file_entity_tag function on page 353
- file_is_graphic function on page 354
- framesets function on page 362
- graphic_attr_name function on page 372
- graphic_entity_attr_name function on page 373
- graphic_entity_names function on page 374
- graphic_entity_tag function on page 374
- graphic_file_attr_name function on page 375
- graphic_format function on page 375
- graphic_information function on page 376
- graphic_relative_path function on page 377
- graphic_tag function on page 377
- graphic_tag_name function on page 377
- hidden_tag function on page 379
- htmldoc function on page 380

- `tag_display (Function)` function on page 538
- `tag_diaplay_name` function on page 539
- `tag_exists` function on page 539
- `tag_has_attr` function on page 540
- `tag_has_conref` function on page 540
- `tag_names` function on page 541
- `tag_names_ns` function on page 541
- `tag_real_name` function on page 542
- `tag_substitutions` function on page 542
- `text_entity_names` function on page 573
- `text_entity_tag` function on page 573
- `text_style_tag_name` function on page 574
- `text_style_tag_name_ns` function on page 574
- `uri_resolve` function on page 582
- `user_tag_names` function on page 584

Dynamic Loading (API) Functions

Dynamic loading functions provide a direct interface to the operating system facilities for loading, calling, and unloading functions in dynamically loaded (shared) libraries. In the Windows environment, this includes the following dynamic link library APIs: `LoadLibrary`, `FreeLibrary` and `GetProcAddress`. These functions also allow you to extend ACL by writing DLLs in C or in other programming languages that support DLLs.

- `dl_builtin_addr` function on page 271
- `dl_call` function on page 272
- `dl_error` function on page 275
- `dl_find` function on page 275
- `dl_load` function on page 276
- `dl_unload` function on page 277
- `dom_address` function on page 338
- `dom_document` function on page 338
- `dom_free` function on page 339
- `dom_oid` function on page 339
- `interpreter_addr` function on page 389

- `pack` function on page 478
- `unpack` function on page 579

Editor Functions

Editor functions query or change the state of a document instance. The OID functions that change the document tree are also found in the “Object identifier (OID) functions” topic.

- `caret_at` function on page 235
- `caret_in_selection` function on page 235
- `caret_show` function on page 235
- `compare_files` function on page 249
- `current_tag_attr_value` function on page 255
- `current_tag_name` function on page 256
- `detail_tag` function on page 265
- `direction` function on page 266
- `doc_from_compare` function on page 319
- `doc_save` function on page 329
- `doc_word_count` function on page 337
- `edit_id` function on page 342
- `edit_new_window` function on page 342
- `find` function on page 359
- `forward_char` function on page 362
- `generate_id` function on page 364
- `get_default_printer` function on page 368
- `get_newlist_entry` function on page 368
- `get_num_printers` function on page 369
- `goto_oid` function on page 372
- `graphic_viewer` function on page 378
- `insert` function on page 384
- `insert string (Function)` function on page 386
- `insert_tag (Function)` function on page 387
- `inside_tag` function on page 388
- `is_postscript_printer` function on page 389

- item_tag function on page 390
- list_tag function on page 404
- looking_at function on page 405
- lookup function on page 406
- lookup_types function on page 406
- lookup_replacements function on page 406
- marking function on page 410
- modified function on page 417
- modify_attr function on page 417
- mouse_at function on page 419
- mouse_in_selection function on page 422
- oid_delete function on page 434
- oid_delete_attr function on page 435
- oid_modify_attr function on page 450
- oid_top_pos function on page 460
- replace function on page 505
- scroll_to_oid function on page 510
- selected function on page 510
- selected_element function on page 511
- selection_anchor function on page 511
- selection_balanced function on page 511
- selection_end function on page 512
- selection_markup function on page 512
- selection_start function on page 513
- spellskip_tag function on page 517
- target_id_attr_name function on page 543
- target_tag function on page 543
- target_tag_name function on page 543
- thesaurus function on page 574

Entity Functions

Entity functions provide information about general text entities or external file entities.

- add_filep_entity function on page 215
- declare_char_entity function on page 261
- declare_file_entity function on page 261
- declare_filep_entity function on page 261
- declare_graphic_entity function on page 262
- declare_notation function on page 263
- declare_text_entity function on page 263
- doc_flatten function on page 318
- entity function on page 343
- entity_doc function on page 343
- entity_exists function on page 343
- entity_expand function on page 344
- entity_first function on page 344
- entity_last function on page 344
- entity_name function on page 344
- entity_notation function on page 345
- entity_parfile function on page 345
- entity_path function on page 345
- entity_pubid function on page 345
- entity_relative_path function on page 346
- entity_resolve function on page 346
- entity_source function on page 347
- entity_sysid function on page 347
- entity_tag function on page 348
- entity_to_unicode function on page 348
- entity_type function on page 348
- file_entity_names function on page 353
- filep_entity_names function on page 358
- graphic_entity_names function on page 374
- insert_filep_entity function on page 386
- modify_file_entity function on page 418
- modify_graphic_entity function on page 418
- modify_text_entity function on page 418

- `msp_entity_names` function on page 422
- `rename_ms_parameter` function on page 505
- `text_entity_names` function on page 573
- `undeclare_ms_parameter` function on page 578
- `unicode_to_entity` function on page 579

File Identifier Functions

File identifier functions manipulate file identifiers (FIDs) returned by the `open` function. These functions provide an interface to the C-library standard I/O routines.

- `close` function on page 244
- `eof` function on page 349
- `flush` function on page 361
- `get_preferences_path` function on page 369
- `getline` function on page 370
- `open` function on page 469
- `put` function on page 501
- `read (Function)` function on page 502
- `seek` function on page 510
- `tell` function on page 572
- `truncate` function on page 577
- `write (Function)` function on page 607

File or System Functions

File or system functions provide file or system information or are used to manipulate pathnames.

- `absolute_file_name` function on page 214
- `access` function on page 214
- `basename` function on page 229
- `dirname` function on page 266
- `disable_windows` function on page 266
- `doc_clean_cache` function on page 313
- `drop_file_info` function on page 340

- `expand_file_name` function on page 352
- `error` function on page 349
- `errors_suppressed` function on page 349
- `file_close` function on page 352
- `file_directory` function on page 353
- `file_mtime` function on page 354
- `file_newer` function on page 355
- `file_public_id` function on page 355
- `file_size` function on page 356
- `file_system` function on page 357
- `filename_to_url` function on page 358
- `find_file_in_path` function on page 360
- `getpid` function on page 371
- `glob` function on page 371
- `graphic_viewer` function on page 378
- `http_cache_flush` function on page 382
- `pwd` function on page 501
- `system` function on page 531
- `temp_name` function on page 572
- `unmask` function on page 577
- `universal_file_name` function on page 579
- `url_decode` function on page 582
- `url_encode` function on page 583
- `username` function on page 584

Hook Functions

A hook is a user-defined function that is called at strategic places by Arbortext Editor so you can customize editing operations. Setting a hook has global effect unless you build in additional user functions to specify a local level.

- `adapterstatehook` function on page 939
- `catalogpathhook` function on page 939
- `changetrackingaccepthook` function on page 940
- `changetrackingafterhook` function on page 940

- preferencehook function on page 966
- preimporthook function on page
- previewlinkhook function on page 967
- printcompletehook function on page 967
- profiledochook function on page 967
- tblmodelprompthook function on page 968
- untrackedchangehook function on page 969
- userulehook function on page 969
- writetexafterhook function on page 972
- writetexhook function on page 973

Import/Export Functions

These functions start either an import or export process with the specified parameters.

- document_export function on page 337

Java, JavaScript, JScript, and VBScript Functions

Java, JavaScript, JScript, and VBScript expressions and scripts can be called, created, read and executed from Arbortext Command Language using the functions listed.

- append_javaclass_path function on page 221
- java_array_from_acl function on page 390
- java_array_to_acl function on page 391
- java_console function on page 392
- java_constructor function on page 393
- java_constructor_modal function on page 393
- java_delete function on page 393
- java_init function on page 394
- java_instance function on page 394
- java_instance_modal function on page 394
- java_static function on page 395
- java_static_modal function on page 395

- `javascript` function on page 396
- `jscript` function on page 397
- `js_source` function on page 397
- `vbscript` function on page 586

Layout Functions

These functions enable changes to be made to page layout.

- `apply` function on page 225
- `linenum` function on page 401
- `oid_find_valid_insert` function on page 441
- `oid_logical_mate` function on page 450

Macro Recorder Functions

These functions are available for customizing a site's use of the macro recorder.

- `macro_exists` function on page 407
- `macro_pause_recording` function on page 407
- `macro_record` function on page 408
- `macro_record_cmd` function on page 408
- `macro_recording` function on page 409
- `macro_run` function on page 409
- `macro_running` function on page 410
- `macro_stop_recording` function on page 410

Menu and Keymap Functions

The menu and keymap functions query keymaps or the menu system.

- `key_cmd` function on page 398
- `keymap_exists` function on page 398
- `menu_active` function on page 413
- `menu_checked` function on page 413
- `menu_cmd` function on page 414
- `menu_exists` function on page 414
- `menu_item_array` function on page 414

- menu_item_count function on page 415
- menu_popup function on page 415
- undo_menu_description function on page 578
- undo_menu_description_clear function on page 578

Message Localization Functions

These functions are used when working with localized message files.

- agettext function on page 217
- amo_close function on page 217
- amo_open function on page 217
- amo_text function on page 218
- cjk_locale function on page 243
- getlocale function on page 371
- locale_file_name function on page 404

Miscellaneous Functions

These are miscellaneous functions for operating within Arbortext Editor.

- exit_editor function on page 352
- font_families function on page 361
- function_argc function on page 363
- function_file function on page 363
- function_names function on page 363
- init_done function on page 384
- license function on page 399
- license_info function on page 400
- license_release function on page 400
- terminal_mode function on page 573

Notation Functions

Notation functions provide information about notations declared in a document.

- notation_exists function on page 423
- notation_names function on page 423

- notation_parfile function on page 423
- notation_pubid function on page 424
- notation_source function on page 424
- notation_sysid function on page 424

Numeric Functions

These functions are used when returning numeric character values.

- chr function on page 242
- dimen_convert function on page 265
- hex function on page 379
- max function on page 411
- min function on page 417
- oct function on page 427
- ord function on page 477

Object Identifier (OID) Functions

An OID is a unique identifier that locates an element within the structure of a parsed document instance. The OID also records to which document instance the element belongs. This means you can save an OID in a variable, and then later refer to it in OID functions. It makes no difference what the current document is.

- doc_first_dobj function on page 318
- doc_next_dobj function on page 325
- goto_oid function on page 372
- oid_asis function on page 427
- oid_attr function on page 428
- oid_attr_list function on page 428
- oid_attr_required function on page 429
- oid_attr_type function on page 429
- oid_backward function on page 429
- oid_caret function on page 430
- oid_caret_offset function on page 430
- oid_caret_pos function on page 431
- oid_check_attr function on page 431

- `oid_select` function on page 456
- `oid_set_graphic_pathname` function on page 456
- `oid_set_icon` function on page 456
- `oid_show_attr` function on page 459
- `oid_split_tag` function on page 459
- `oid_tbl_obj_list` function on page 459
- `oid_top_pos` function on page 460
- `oid_treeloc` function on page 460
- `oid_type` function on page 461
- `oid_unknown` function on page 461
- `oid_unknown_attr_list` function on page 462
- `oid_valid` function on page 462
- `oid_visible_change_tracking` function on page 462
- `oid_write_graphic` function on page 463
- `oid_xpath_boolean` function on page 465
- `oid_xpath_integer` function on page 466
- `oid_xpath_matches` function on page 467
- `oid_xpath_nodeset` function on page 468
- `oid_xpath_string` function on page 468
- `scroll_to_oid` function on page 510
- `selected_element` function on page 511
- `selection_end` function on page 512
- `selection_start` function on page 513
- `treeloc_oid` function on page 576

Profiling Functions

These functions support profiling and allow for site-specific customizations of Arbortext Editor profiling capabilities.

Profile Functions

- `profile_alias` function on page 484
- `profile_aliases` function on page 484
- `profile_allowed` function on page 484

- profile_attr function on page 484
- profile_attrs function on page 485
- profile_class_values function on page 485
- profile_classes function on page 485
- profile_config function on page 486
- profile_conflictshadingbackground function on page 486
- profile_default_value function on page 487
- profile_default_value_node function on page 487
- profile_element_allowed function on page 487
- profile_element_attr_tests function on page 487
- profile_elements_list function on page 488
- profile_is_foldered function on page 488
- profile_is_radiochoice function on page 488
- profile_is_standard function on page 489
- profile_resolution function on page 489
- profile_rootnode function on page 489
- profile_rootnodes function on page 489
- profile_shadingbackground function on page 486
- profile_type function on page 490
- profile_valid function on page 490
- profile_value_node function on page 491
- profile_value_nodes function on page 491
- profile_value_separator function on page 491
- profile_values function on page 492
- profile_values_shadingbackground function on page 492

Profile Group Functions

- apply_profile_group function on page 226
- apply_profile_group_allowed function on page 226
- apply_profile_group_value_nodes function on page 226
- apply_profile_groups function on page 227
- set_profile_group function on page 514

- `set_profile_groups` function on page 514
- `set_profile_groups_expressions` function on page 515

Profile Node Functions

- `profilenode_ancestors` function on page 492
- `profilenode_attr` function on page 493
- `profilenode_children_nodes` function on page 493
- `profilenode_default_value` function on page 493
- `profilenode_default_value_node` function on page 493
- `profilenode_element_allowed` function on page 494
- `profilenode_element_attr_tests` function on page 494
- `profilenode_elements_list` function on page 494
- `profilenode_is_foldered` function on page 495
- `profilenode_is_radiochoice` function on page 495
- `profilenode_is_standard` function on page 495
- `profilenode_name` function on page 496
- `profilenode_parent` function on page 496
- `profilenode_rootnode` function on page 496
- `profilenode_shadingbackground` function on page 496
- `profilenode_type` function on page 497
- `profilenode_valid` function on page 497
- `profilenode_valid_nodes` function on page 497
- `profilenode_value_separator` function on page 497
- `profilenode_values` function on page 498

Schema Functions

These functions validate a document against a schema.

- `ns_schema_validate_batch` function on page 425
- `schema_validate` function on page 508
- `schema_validate_batch` function on page 509

String Functions

A string specifies the text to be inserted at the cursor location.

- `chop` function on page 242
- `chr` function on page 242
- `dupl` function on page 342
- `gsub` function on page 378
- `hex` function on page 379
- `index` function on page 384
- `join (Function)` function on page 396
- `length` function on page 399
- `match` function on page 411
- `match_length` function on page 411
- `match_result` function on page 411
- `match_start` function on page 411
- `oct` function on page 427
- `ord` function on page 477
- `reverse` function on page 507
- `rindex` function on page 507
- `span` function on page 517
- `split (Function)` function on page 518
- `strcoll` function on page 518
- `sub` function on page 530
- `substr` function on page 531
- `tolower` function on page 576
- `toupper` function on page 576
- `trim` function on page 577

Set Option Functions

Set options define how a scope can affect your Arbortext Editor session. These functions either return or set the value of the specified option, which is used to update the current scope.

- `doc_get` function on page 320
- `doc_set` function on page 330

- `languages` function on page 398
- `option_scope` function on page 475
- `option_type` function on page 475
- `option_value_max` function on page 476
- `option_value_min` function on page 476
- `option_value_units` function on page 477
- `option_value_validate` function on page 477
- `option_values` function on page 477
- `preference` function on page 483
- `read_preferences` function on page 503
- `window_reset_configuration` function on page 597
- `window_set` function on page 597
- `write_preferences` function on page 608

Stylesheet Functions

Stylesheets are a collection of style settings that control the appearance of a document. These functions control operations within the stylesheet.

- `clear_stylesheet` function on page 243
- `fosi_public_id` function on page 362
- `list_stylesheets` function on page 404
- `styler` function on page 518
- `styler_enabled` function on page 519
- `styler_get_styled_elements` function on page 519
- `stylesheet` function on page 520
- `stylesheet_export_fosi` function on page 520
- `stylesheet_export_xsl` function on page 520
- `stylesheet_gentext_lang_stats` function on page 521
- `stylesheet_get_list_dir` function on page 523
- `stylesheet_get_list_doc` function on page 524
- `stylesheet_import_xlf` function on page 525
- `stylesheet_list_add` function on page 527
- `stylesheet_new` function on page 528
- `stylesheet_revert` function on page 528

- stylesheet_save function on page 528
- stylesheet_saveas function on page 529
- stylesheet_select function on page 529

Table Functions

Table functions edit tables in the Edit window.

- go_to_cell function on page 372
- tbl_area_celllist function on page 544
- tbl_caret_col function on page 544
- tbl_caret_row function on page 544
- tbl_dlg_target function on page 549
- tbl_edit_close function on page 549
- tbl_edit_open function on page 549
- tbl_hline_rulelist function on page 552
- tbl_insert function on page 553
- tbl_insert_rows_cols_dlg function on page 553
- tbl_insert_table_dlg function on page 553
- tbl_insertion_valid function on page 554
- tbl_mod_borders_dlg function on page 554
- tbl_mod_cellfont_dlg function on page 554
- tbl_mod_cells_dlg function on page 555
- tbl_mod_table_dlg function on page 555
- tbl_model_celllist function on page 555
- tbl_model_id function on page 555
- tbl_model_list function on page 555
- tbl_model_name function on page 556
- tbl_model_operation function on page 556
- tbl_model_row function on page 558
- tbl_model_support function on page 559
- tbl_model_table_title function on page 559
- tbl_model_tablelist function on page 559
- tbl_model_taglist function on page 559
- tbl_model_wrapperlist function on page 560

- `tbl_multicell_border` function on page 560
- `tbl_multicell_celllist` function on page 560
- `tbl_multicell_neighborlist` function on page 560
- `tbl_multicell_size` function on page 560
- `tbl_multicell_spanner` function on page 561
- `tbl_selection_clone` function on page 568
- `tbl_selection_empty` function on page 569
- `tbl_selection_get` function on page 569
- `tbl_selection_matchbegin` function on page 569
- `tbl_selection_matchnext` function on page 569
- `tbl_selection_nextrectangle` function on page 570
- `tbl_selection_pasterectangle` function on page 570
- `tbl_selection_pastetype` function on page 570
- `tbl_selection_restore` function on page 571
- `tbl_selection_tmid` function on page 571
- `tbl_selection_valid` function on page 571
- `tbl_table_title_delete` function on page 572
- `tbl_table_titlle_insert` function on page 572
- `tbline_vline_rulelist` function on page 572
- `window_cur_table` function on page 591

Table Object Attributes

Attribute values are initially derived from table markup. Values may be manipulated by the following ACL functions:

- tbl_obj_attr_clear on page 562
- tbl_obj_attr_delete on page 562
- tbl_obj_attr_get on page 562
- tbl_obj_attr_set on page 563

Some values are numbers, some values are strings. String values are case insensitive.

Shared Table Attributes

The following attributes are supported on all table objects (sets, grids, columns, rows, cells, and rules).

OID

This attribute identifies the markup tag corresponding to the set, grid, row, column, cell or rule. Depending upon the table model, this value may be null. For example, there's no tag corresponding to a column under the Arbortext table model.

The user is not allowed to set this attribute.

RESIZABLE

This attribute is zero (`0`) for not resizable or one (`1`) for resizable. It is always zero (`0`) for a set, cell or rule, as these items are not considered resizable. For a grid, row, or column, this value indicates whether the width (grid or column) or height (row) may be changed. The value will be zero (`0`) if the relevant parts of the document are read-only or if a variety of other internal conditions apply.

The user is not allowed to set this attribute.

Set Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attributes are supported on set table objects.

CELLPADDING

This attribute applies to HTML tables only.

This attribute controls the space between cell wall and the cell contents (the cell wall is at the center of the cell border, not at its edge). The only supported unit is `px` (pixel). Enter a whole number value in the field provided. The default value is `2` pixels.

CELLSPACING

This attribute applies to HTML tables only.

This attribute controls the space between cells and between the table outer border and the table cells. The only supported unit is `px` (pixel). Enter a whole number value in the field provided. The default value is `1` pixel.

DEFAULT_RTH_UNIT

This attribute indicates the default “row target height” unit. If a row has an explicit height, for example `1.0in`, and the user drag-resizes the row's height, we store the new height to markup using `in` (inches) as the unit.

Consider the case, however, in which a table has no explicit row heights given in its markup. If the user drag-resizes a row, Arbortext Editor wants to store the value thus created, but doesn't know which unit to use. This attribute provides an answer to the question.

If a row has no explicit height set and a user's edit forces Arbortext Editor to pick a unit, it uses the unit named by this attribute on the Set containing the Grid containing the Row.

Permissible units in OASIS Exchange and Arbortext tables are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, the only allowable row height unit is pixels. Consequently, this attribute is not supported in HTML tables.

FRAME

This attribute indicates how the outer borders of a Set are drawn. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Value	Effect
`top`	Draws the top border of the set only.
`bottom`	Draws the bottom border of the set only.
`topbot`	Draws the top and bottom borders of the set only.
`all`	Draws all four borders of the set.
`sides`	Draws the left and right borders of the set only.
`left`	Draws the left border of the set only.*
`right`	Draws the right border of the set only.*
`none`	Draws no borders.

*Applies to HTML tables only.

FRAME_COLOR_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the top, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the bottom, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the left, outer border. The value is either a named color or an RGB value.

FRAME_COLOR_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the color of the right, outer border. The value is either a named color or an RGB value.

FRAME_STYLE_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the top, outer border. The following values are supported:

- `blank`
- `single`
- `double`
- `dashed`
- `dotted`

FRAME_STYLE_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the bottom, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_STYLE_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the left, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_STYLE_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the style of the right, outer border. The supported values are the same as the FRAME_STYLE_ABOVE attribute.

FRAME_WIDTH_ABOVE

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the top, outer border. The value is a decimal number followed by a unit of measure, for example `1.0pt`. The following units of measurement are supported:

- `pt` — points
- `pi` — picas
- `in` — inches
- `mm` — millimeters
- `cm` — centimeters

- `px` — pixels.
- `em` — em space

FRAME_WIDTH_BELOW

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the bottom, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAME_WIDTH_LEFT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the left, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAME_WIDTH_RIGHT

This attribute applies to OASIS, HTML, and custom tables.

This attribute controls the width of the right, outer border. The supported values are the same as the FRAME_WIDTH_ABOVE attribute.

FRAMERULES

This attribute applies to custom tables only.

This attribute sets the rules that are drawn for the custom table. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Value	Effect
`all`	Draws all rules. (the default)
`top`	Draws the top frame rule only.
`bottom`	Draws the bottom frame rule only.
`topbot`	Draws the top and bottom frame rules only.
`left`	Draws the left frame rules only.
`right`	Draws the right frame rules only.
`sides`	Draws the left and right frame rules only.
`rows`	Draws the horizontal rules only.
`cols`	Draws the vertical rules only.
`frame-only`	Draws the outer frame rules only.
`none`	Draws no rules.

FRAMETHICKNESS

This attribute applies to HTML tables only.

This attribute sets the width of the outer table border. The only supported unit is `px` (pixels). Attribute values must be whole numbers.

GUITYPE

This attribute returns the string `oasis`, `ati`, or `html` to specify that the table editor is to use the Oasis Exchange, Arbortext, or HTML (respectively) dialog box options for the table.

The user is not allowed to set this attribute.

RULE

This attribute applies to HTML tables only.

This attribute sets which rules are drawn for the inside of the table. The outer borders are controlled by the `FRAME` attribute. Note that the setting of this attribute is overridden by any style setting for an individual frame or inner rule. The following values are supported:

Supported values for `RULE` attribute

`all`	Draws all the internal rules within the set.
`rows`	Draws only the horizontal rules between the rows within the set.
`cols`	Draws only the vertical rules between the columns within the set.
`groups`	Draws a single horizontal rule between the header and body rows and another single horizontal rule between the body and footer rows.
`none`	Draws no rules within the set.

TMID

This attribute is the Table Model ID of the Table Model that is managing the Set and its content.

The user is not allowed to set this attribute.

Grid Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attributes are supported on grid table objects.

RULETHICKNESS

This attribute is a unitdim value indicating the thickness of the rules. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

USEWIDTH
This attribute is zero (`0`) or one (`1`), indicating whether the Grid's `WIDTH` attribute should be ignored (`0`) or used (`1`). While the user cannot set this attribute directly, if they explicitly set the `WIDTH` attribute, this attribute's value automatically changes to one (`1`). If the user resets the `WIDTH` attribute to the default value, this attribute value automatically changes to zero (`0`).

WIDTH
If the `USEWIDTH` attribute is a one (`1`), this attribute is a unitdim indicating the width of the Grid on the screen. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

> **Note**
> This attribute (or, more precisely, the markup underlying it) is not used in published output.

Column Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attributes are supported on column table objects.

WIDTH_MAX
This attribute is a unitdim indicating the maximum width to which the column can be drag-resized. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

WIDTH_MIN
This attribute is a unitdim indicating the minimum width to which the column can be drag-resized. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

COLWIDTH
This attribute is a unitdim indicating the column's width. In addition to the units allowed for other unitdims (that is, `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters), the column's width unit may also be `*`, indicating proportional width units.

WIDTH_UNIT
This attribute is the unit portion of the `COLWIDTH` attribute. This attribute exists to allow the user to convert between units without doing actual calculations. If a column's `COLWIDTH` attribute is `1.0in`, a call to the `tbl_obj_attr_set` function with a `WIDTH_UNIT` attribute value of `cm` will cause the column's `COLWIDTH` unit to be changed to `2.54cm`.

Row Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attributes are supported on row table objects.

BREAK_PENALTY

The value of `BREAK_PENALTY` is used to determine whether to cause a page break. `BREAK_PENALTY` can be one of the following values:

- `default` — No preference on a break.
- `allow` — Permit a break.
- `inhibit` — Do not permit a break.
- `force` — Insert a break.
- `recto` — Break to the next odd page.
- `verso` — Break to the next even page.

FOOTER_LEVEL

If the row is a footer row, this attribute will be one (`1`) otherwise it will be zero (`0`). This attribute cannot be set to one (`1`) if the `HEADER_LEVEL` attribute is already set to one (`1`).

For the Arbortext table model, no row may have a non-zero footer level. For the CALS table model (OASIS Exchange), the table can include a block of header rows, followed by a block of footer rows, followed by a block of body rows. For the CALS table model, setting a row's footer level will cause the row to be moved to the footer or body block, as appropriate.

You can not mark a row as a footer row if doing so removes the only row from the table that is not a footer row. For example, if there is only one row in the table, it can't be a footer row as there must always be at least one row in the table that is not a footer row.

Do not change the value of `FOOTER_LEVEL`. Doing so will render the original row invalid and may cause an AOM exception to be thrown or cause Arbortext Editor to terminate unexpectedly.

HEADER_LEVEL

If the row is a header row, this attribute will be one (`1`), otherwise it will be zero (`0`). This attribute cannot be set to one (`1`) if the `FOOTER_LEVEL` attribute is already set to one (`1`).

For the Arbortext table model, only the first row in the table may have a nonzero header level. For the CALS table model (OASIS Exchange), the table can include of a block of header rows, followed by a block of footer rows, followed by a block of body rows. For the CALS table model, setting a row's header level will cause the row to be moved to the header or body block, as appropriate.

You can't mark a row as a header row if this action removes the only row from the table that is not a header row. For example, if there is only one row in the table, it can't be a header row as there must always be at least one row in the table that is not a header row.

Do not change the value of `HEADER_LEVEL`. Doing so will render the original row invalid and may cause an AOM exception to be thrown or cause Arbortext Editor to terminate unexpectedly.

HEIGHT_UNIT

This attribute is the unit portion of the `TARGET_HEIGHT` attribute. This attribute exists to allow the user to convert between units without doing actual calculations. If a row's `TARGET_HEIGHT` attribute is `1.0in`, a call to the `tbl_obj_attr_set` function with a `HEIGHT_UNIT` attribute value of `cm` will cause the row's `TARGET_HEIGHT`unit to be changed to `2.54cm`.

TARGET_HEIGHT

If the row's `USE_TARGET_HEIGHT` attribute is one (`1`), this attribute is a unitdim value indicating the height that is used to display the row. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

Setting this attribute causes the `USE_TARGET_HEIGHT` attribute to be set to one (`1`). Resetting this attribute to its default value (using `tbl_obj_attr_delete`) causes the `USE_TARGET_HEIGHT` attribute to be set to zero (`0`).

USE_TARGET_HEIGHT

This attribute may be zero (`0`) or one (`1`) to indicate whether the value of the `TARGET_HEIGHT` attribute should be used. The user can not set this attribute directly, but can influence its value by calling `tbl_obj_attr_get` or `tbl_obj_attr_delete` for the `TARGET_HEIGHT` attribute.

Cell Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attributes are supported on cell table objects.

BGCOLOR

This attribute controls the background color of the cell. The value is either a named color or an RGB value.

BOTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's bottom rule and the edge of the box in which text is actually displayed. Permissible values are `in`, `pt`, `pi`, `px`, `cm`, or `mm` meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (px is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the CELLPADDING set table attribute.

LEFTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's left rule and the edge of the box in which text is actually displayed. Permissible values are in, pt, pi, px, cm, or mm meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (px is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the CELLPADDING set table attribute.

RIGHTMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's right rule and the edge of the box in which text is actually displayed. Permissible values are in, pt, pi, px, cm, or mm meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (px is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the CELLPADDING set table attribute.

TOPMARGIN

This attribute has a unitdim value indicating the amount of white space to be left between a cell's top rule and the edge of the box in which text is actually displayed. Permissible values are in, pt, pi, px, cm, or mm meaning inches, points, picas, pixels, centimeters, or millimeters.

In HTML tables, setting this attribute (px is the only supported unit) will set all four margin attributes for all the cells in the table. You can achieve the same effect by setting the CELLPADDING set table attribute.

HALIGN

This attribute controls horizontal text justification for the cell. Permissible values are left, center, right, justified, or character.

The following values are also valid, but cannot be used to control the horizontal text justification for the cell: start, end, or inherit.

Character alignment is not effective in HTML outputs. Browsers are not required to support it, and they do not.

HALIGN_CHAR

This attribute is a single character on which to horizontally align the text in a column of cells. This character alignment scheme is used only if the HALIGN attribute is set to character.

HALIGN_CHAR_OFF
This attribute is a number between zero (0) and 100 indicating the desired location of the alignment character given by the HALIGN_CHAR attribute. Zero (0) places the character to the extreme left side of the cell, 100 places the character to the extreme right of the cell. This offset is used only if the HALIGN attribute is set to character.

HEADER
This attribute applies only to HTML tables. This attribute is 0 if the cell is a data cell (td) and is 1 if the cell is a header cell (th).

RULECOLOR_ABOVE
This is a convenience attribute to allow the COLOR attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_BELOW
This is a convenience attribute to allow the COLOR attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_LEFT
This is a convenience attribute to allow the COLOR attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_RIGHT
This is a convenience attribute to allow the COLOR attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the COLOR attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_ABOVE
This is a convenience attribute to allow the STYLE attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_BELOW
This is a convenience attribute to allow the STYLE attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_LEFT
This is a convenience attribute to allow the STYLE attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULESTYLE_RIGHT
This is a convenience attribute to allow the `STYLE` attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the STYLE attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_ABOVE
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule at the top of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_BELOW
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule at the bottom of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULEWIDTH_LEFT
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule on the left side of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

RULECOLOR_RIGHT
This is a convenience attribute to allow the `RULEWIDTH` attribute of the rule on the right side of the cell to be manipulated. Refer to the explanation of the RULEWIDTH attribute in the Rule table attributes on page 135 help topic.

VALIGN
This attribute is either `top`, `center`, or `bottom` to indicate vertical alignment of the contents of the cell.

Rule Table Attributes

In addition to the attributes listed in the Shared table attributes on page 177 help topic, the following attribute is supported on rule table objects.

COLOR
This attribute indicates the color of the rule should be drawn. The value is either a named color or an RGB value.

RULEWIDTH
This attribute indicates the width of the rule. The value is a decimal number followed by a unit of measure, for example `1.0pt`. The following units of measurement are supported:

- `pt` — points
- `pi` — picas
- `in` — inches

- `mm` — millimeters
- `cm` — centimeters
- `px` — pixels.
- `em` — em space

STYLE
This attribute indicates how the rule should be drawn. The following values are supported:

- `blank`
- `single`
- `double`
- `dashed`
- `dotted`

Time Functions

Time functions return time information or create timers.

- `ctime` function on page 254
- `elapsed_time` function on page 343
- `file_mtime` function on page 354
- `time_date` function on page 575
- `timer_add_callback` function on page 1026
- `timer_remove_callback` function on page 1027
- `time (Function)` function on page 575
- `times` function on page 576

Translation Functions

These functions find, return , and set values of a translated version of a document.

- `doc_is_translation` function on page 323
- `doc_get_translation_locale` function on page 321
- `doc_set_translation_locale` function on page 332
- `doc_get_translation_orig_uri` function on page 321
- `doc_set_translation_orig_uri` function on page 332

Window Functions

Window functions operate on window identifiers.

- window_reset_configuration function on page 597
- window_set function on page 597
- window_set_columnview function on page 603
- window_show function on page 604
- window_split function on page 604
- window_state function on page 604
- window_sync function on page 604
- window_sync_pane function on page 605
- window_sys_close function on page 605
- window_sys_keymenu function on page 605
- window_sys_maximize function on page 605
- window_sys_minimize function on page 605
- window_sys_move function on page 606
- window_sys_restore function on page 606
- window_sys_size function on page 606
- window_table_left_column function on page 606
- window_table_right_column function on page 606
- window_update function on page 606
- window_xid function on page 607
- windows_disabled function on page 607

XPath Functions

Placeholder for introductory content.

- xpath_boolean function on page 608
- xpath_integer function on page 609
- xpath_nodeset function on page 610
- xpath_string function on page 611
- xpath_valid function on page 611

4

Functions by Alphabetical Listing

absolute_file_name

absolute_file_name (*filename*[, *dir*])

This function returns *filename* converted to an absolute path name. If *filename* is relative, the function prepends the directory *dir* if that directory is given; otherwise, the current working directory is used to make *filename* absolute.

If *filename* starts with ~ or contains any variable references, the name is expanded by calling `expand_file_name` on page 352. If *filename* contains the `%L` symbolic parameter, the name is expanded by calling `locale_file_name` on page 404. Arbortext Editor simplifies file names containing `.` or `..` as components.

access

access(*pathname*, *string*)

This function returns `1` (True) if the file name specified by the expression *pathname* is accessible according to the mode given by the string *string*.

The *pathname* argument is either a file system path or an HTTP or HTTPS resource. For the latter, the function issues an HTTP HEAD request and uses the response to that request to determine if the resource can be accessed.

The *string* argument is either `e` (for exists), or a concatenation of one or more of the strings `r` (for “read”), `w` (for “write”), and `x` (for “executable”) based on system permissions for the file. For example:

```
if (access("doc.sgm", "rw"))
```

tests for both read and write permission to the file `doc.sgm`. If the file does not exist, or if it does not have each of the permissions specified, `0` is returned.

add

layout::add ([*flags*[, *file*]])

This function declares the `atipl` namespace, adds page layout markup to a document, and returns the status of the operation. The added markup consists of `atipl` singletons. The *flags* parameter is a bit mask used to control the markup inserted. The *file* parameter specifies the location of the `layout` file that drives the operation. This `layout` file is generated automatically when line numbering is applied to a document.

If the operation fails because the source document was write protected or missing, or the layout file is not found or improper, then this function returns `-1`. If the document is modified but some markup could not be inserted in the required location, then this function returns `0`. Otherwise, the call is successful and this function returns `1`.

The failure to insert some layout tags in the appropriate locations is expected in some applications (for example, CALS change pages with generated text, tables of contents, and so on). In these cases the `atipl` tag is inserted at the nearest previous location where it is legal to do so, and the *error* attribute of the markup is set to `1`.

The flag bits control the type of structure markup to add to the document. These bits are:

- `0x1` — *line*
- `0x2` — *entry*
- `0x4` — *row*
- `0x8` — *float*
- `0x10` — *column*
- `0x20` — *span*
- `0x40` — *page*

The default behavior when *flags* is not specified is to output all structures.

For more detailed information on the page layout elements and their attributes, see http://www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

add_filep_entity

add_filep_entity (*name*[, *sysid*[, *pubid*]])

This function allows you to declare a file parameter entity (that is, an external parameter entity) in the document's internal subset and insert a reference to the parameter entity.

name is the name for the entity. A leading percent sign `%` is optional.

sysid is the entity's system identifier.

pubid is the entity's public identifier.

For example, after this function call:

```
add_filep_entity("myent", "/usr/myent")
```

the document's internal declaration will contain:

```
<!ENTITY % myent SYSTEM "/usr/myent">
%myent;
```

The function call in the example has the same effect as these two function calls:

```
declare_filep_entity("myent", "/usr/myent")
insert_filep_entity("myent")
```

Note

To ensure the file parameter's declarations are added to the document's declaration, save the document, then choose **File ► Revert to Saved**.

add_graphic_fallback

add_graphic_fallback (*primary*, *fallbacklist*, *isPrint*)

This function allows you to declare fallback types for graphic formats not natively supported by Arbortext Editor.

primary is the file extension for the primary graphic type without a dot, for example *c3di* or *cnv*.

fallbacklist is a comma separated list defining the file extensions to look for in secondary (attachment) content

isPrint is the output type for which the fallback types should apply — *1* for printed output, *0* for screen (online) outputs

For example:

```
add_graphic_fallback("c3di", "pvz,tif", 0)
```

For Arbortext Editor connected via the PTC Server connection, the fallback uses the adapter call `sess_get_file()` to obtain the fallback rendition.

When publishing a payload with Arbortext Publishing Engine, the fallback uses the `ResourceMap` in the `manifest.xml` to determine what attachment to use.

If the fallback graphic is not supplied, or there are no matching attachments, publishing will produce an error message

add_hook

add_hook (*hookname*, *funcname*[, *prepend*])

This function adds a callback function to the list of functions for the specified hook, where *hookname* is the name of the hook `set` option. The argument *funcname* is a string specifying the name of the user-defined function to call. If the optional argument *prepend* is specified and non-zero, *funcname* is added to the head of the hook function list. If not specified or zero, *funcname* is appended to the list

`add_hook` will not add if it is already present in the hook list.

Refer to Hook Functions Introduction on page 115 for a detailed introduction to ACL hooks.

agettext

`msg_string=`**`agettext`** `(`*`msg_id`*`)`

This function retrieves the message *msg_id* from the default message file.

Note

Use `amo_text` to retrieve a localized message from a user-defined message file.

If the function executes correctly, it returns the string for the desired message ID. If the function fails, it returns the value of the *msg_id* argument.

Example

```
msg_string = agettext('Context checking is disabled')
```

amo_close

`amo_close` `(`*`msg_file_id`*`)`

This function closes the message file specified by the argument *msg_file_id*. The *msg_file_id* is the message file ID for a message file previously opened with `amo_open`.

If the function executes correctly, it returns the message file ID of the closed file. If the function fails, it returns a -1.

Example

```
$ret = amo_close(msg_file_id)
```

amo_open

`msg_file_id=`**`amo_open`** `(`*`path`*`)`

This function opens the message file specified by the *path* parameter. The *path* parameter should include both a path and file name (including extension) for the desired message file. You must use UNIX path separators (that is, `/`) in your *path* argument.

If the function executes correctly, it returns a message file ID. If the function fails, it returns a -1.

Examples

```
msg_file_id = amo_open('/editor/lib/locale/fr/message.amo')
msg_file_id = amo_open('../custom_messages/message.amo')
```

amo_text

msg_string=**amo_text** (*msg_file_id*, *msg_id*)

This function retrieves the localized message *msg_id* within the message file specified by the *msg_file_id* parameter. The *msg_file_id* parameter is the message file ID for a message file previously opened with amo_open. The *msg_id* parameter is the message ID for a message within the open message file.

Note

Use agettext to retrieve a message from the default message file.

If the function executes correctly, it returns the string for the desired message ID. If the function fails, it returns the value of the *msg_id* parameter.

Example

```
msg_string = amo_text(msg_file_id, \
'Context checking is disabled')
```

append_catalog_path

append_catalog_path (*dir*[, *prepend*])

This function appends the directory *dir* to the set catalogpath option if not already present. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you wanted to append your document types stored in the doctypes subdirectory of the company directory, so your document types are searched last:

```
append_catalog_path('/company/doctypes')
```

If you wanted to prepend your document types stored in that same doctypes subdirectory, so your document types are searched first:

```
append_catalog_path('/company/doctypes',1)
```

If there is a catalog file in the *Arbortext-path*\custom\doctypes subdirectory at startup, the \custom\doctypes path is automatically prepended to the catalog path. If there are any subdirectories of the \doctypes directory that contain a catalog file, those subdirectories are also prepended to the catalog path. Putting a catalog file in \custom\doctypes or in subdirectories of it makes them automatically available, avoiding additional steps to add them to the path.

append_composer_path

append_composer_path (*dir*[, *prepend*])

The append_composer_path function appends *dir* to the directories specified by the set composerpath command. If the optional *prepend* argument is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you want to append a directory stored in the composer subdirectory of the company directory, so your configuration files are searched last:

```
append_composer_path("/company/composer")
```

If you want to prepend your directory stored in the company subdirectory, so your configuration files are searched first:

```
append_copmposer_path('/company/composer',1)
```

If there is an *Arbortext-path*\custom\composer subdirectory at startup, the \custom\composer path is automatically prepended to the path for .ccf files. Putting your .ccf files in the \custom\composer subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_dialogs_path

append_dialogs_path (*dir*[, *prepend*])

This function appends the search path for dialog files that can be called from a custom application, such as one that uses the AOM Application.createDialogFromFile method (documented in the *Programmer's Reference*. The append_dialogs_path function appends *dir* to the directories specified by the set dialogspath command on page 773. If the optional *prepend* argument is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you want to append your dialog file directory stored in the mydialogs subdirectory of the company directory, so your dialog file path is searched last:

```
append_dialogs_path("/company/mydialogs")
```

If you want to prepend your dialog files directory, so your dialog file path is searched first:

```
append_dialogs_path('/company/mydialogs',1)
```

If there is an *Arbortext-path*\custom\dialogs subdirectory at startup, the custom\dialogs path is automatically prepended to the path for dialog files. Putting your custom dialog files in the custom\dialogs subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_dita_path

append_dita_path (*dir*[, *prepend*])

This function appends the directory *dir* to the set ditapath option on page 780 to search for content referenced by DITA documents, if the directory is not already present. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you wanted to append your DITA references directory stored in the ditareferences subdirectory of the company directory, so your DITA references are searched last:

```
append_dita_path("/company/ditareferences")
```

If you wanted to prepend your ditareferences subdirectory, so your DITA references are searched first:

```
append_dita_path('/company/ditareferences',1)
```

If there is an *Arbortext-path*\custom\ditarefs subdirectory at startup, the custom\ditarefs path is automatically prepended to the DITA references path. Putting your DITA references in the custom\ditarefs subdirectory makes them automatically available, avoiding manual steps to add them to the path.

Graphic references in DITA files are resolved using the graphics path.

append_entity_path

append_entity_path (*dir*[, *prepend*])

This function appends the directory *dir* to the set entitypath on page 791 option if not already present. If *prepend* is 1, prepends *dir* to the *entitypath* path, removing any later occurrence of the directory.

Examples

If you wanted to append entities in the /company/entities subdirectory, so your entities are searched last:

```
append_entity_path('/company/entities')
```

If you wanted to prepend entities in the same /company/entities subdirectory, so your entities are searched first:

```
append_entity_path('/company/entities',1)
```

If there is an *Arbortext-path*\custom\entities subdirectory at startup, the \custom\entities path is automatically prepended to the entities path. Putting your entity files in the \custom\entities subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_frameset_path

append_frameset_path (*dir*[, *prepend*])

This function appends the directory *dir* to the set framesetpath on page 814 option if not already present. If *prepend* is 1, prepends *dir* to the *framesetpath* path, removing any later occurrence of the directory.

If there is an *Arbortext-path*\custom\framesets subdirectory at startup, the \custom\framesets path is automatically prepended to the frameset path. Putting your entity files in the \custom\framesets subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_graphics_path

append_graphics_path (*dir*[, *prepend*])

This function appends the directory *dir* to the set graphicspath option if not already present. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you wanted to append your graphics directory stored in the graphics subdirectory of the company directory, so your graphics are searched last:

```
append_graphics_path("/company/graphics")
```

If you wanted to prepend your graphics directory stored in the graphics subdirectory, so your graphics are searched first:

```
append_graphics_path('/company/graphics',1)
```

If there is an *Arbortext-path*\custom\graphics subdirectory at startup, the \custom\graphics path is automatically prepended to the graphics path. Putting your graphics in the \custom\graphics subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_javaclass_path

append_javaclass_path(*dir*[, *prepend*])

This function appends the directory *dir* to the `set javaclasspath` option if not already present. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

Arbortext Editor has an embedded Java Virtual Machine that uses this directory list to search for Java classes when a Java method is called. Arbortext Editor automatically searches the distributed JAR files in the *`Arbortext-path`*`/lib/classes` directory.

Note

Don't place custom JAR files in *`Arbortext-path`*`\lib\classes`. They will be ignored.

Note that any specification for Java class path is only evaluated when the first `java_`*`type`* function is called in a Arbortext Editor session. Subsequent changes to the path will not affect the running Java Virtual Machine until you exit Arbortext Editor and start a new session. Consequently, when using the `java_`*`type`* functions, you should call the `append_javaclass_path` function before invoking a `java_`*`type`* function.

Examples

If you wanted to append your directory for your Java class files stored in the `/company/javaclass` subdirectory, where your Java class directory would be searched last:

```
append_javaclass_path("/company/javaclass")
```

If you wanted to prepend your directory for your Java class files stored in the `/company/javaclass` subdirectory, where your Java class directory would be searched first:

```
append_javaclass_path('/company/javaclass',1)
```

If there is an *`Arbortext-path`*`\custom\classes` subdirectory at startup containing any JAR files (`.jar`), the path for each `.jar` file is automatically prepended to the Java class path for Arbortext Editor. Then the `\custom\classes` path is prepended, which automatically includes any compiled Java `.class` files in it. Putting your `.class` and `.jar` files in the `\custom\classes` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_load_path

`append_load_path` (*`dir`*`[,` *`prepend`*`]`)

This function appends the directory *dir* to the `set loadpath` option if not already present. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

If there is an *Arbortext-path*`\custom\scripts` subdirectory at startup, the `\custom\scripts` is automatically prepended to the load path for ACL, JavaScript, JScript, and VBScript files. Putting these supported script file types in the `\custom\scripts` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_newlist_path

append_newlist_path (*dir*[, *prepend*])

This function appends *dir* to, or removes *dir* from, the `set newlist` command path. *dir* can be a directory path, a directory path and a file name, or a directory path relative to a directory in the catalog path. If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

The `set newlist` command specifies a list of paths to document types that appear in the **Type** list in the **New Document** dialog box.

For example, if you want to append your document type stored in the `c:\mydoctypes\guide` directory, you would type:

```
append_newlist_path('c:\mydoctypes\guide')
```

You can remove a document type from the **Type** list in the **New Document** dialog box. To do this, add a caret (^) character before the document type that you want to remove. You only have to specify the base name of the `.dtd` file.

For example, if you want to remove the Arbortext Editor XML DocBook V4.0 document type from the list, you would type:

```
append_newlist_path('^axdocbook')
```

where `axdocbook` is the base name of the DTD (`axdocbook.dtd`).

If any document types are automatically loaded from the *Arbortext-path*`\custom\doctypes` subdirectory, their paths are also prepended to the document type paths that appear in the **Type** list in the **New Document** dialog box.

append_userdict_path

append_userdict_path (*dir*[, *prepend*])

This function appends the directory *dir* to the `set userdictpath` on page 920 option if not already present. If *prepend* is `1`, prepends *dir* to the user dictionary path, removing any later occurrence of the directory.

If there is an *`Arbortext-path`*`\custom\dictionaries` subdirectory at startup, the `custom\dictionaries` path is automatically prepended to the user dictionary path. Putting your user dictionary files in the `custom\dictionaries` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

append_path

`append_path` (*`pathlist`*`,` *`dir`*`[,` *`prepend`*`]`)

This function returns a string formed by concatenating the path list specified by *pathlist* and the directory specified by *dir* separated by the path list separator $PLS (a semicolon). If the optional argument *prepend* is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, *pathlist* is returned.

append_tagtemplate_path

`append_tagtemplate_path`(*`dir`*`[,` *`prepend`*`]`)

This function appends the directory specified by *dir* to the `set tagtemplatepath` on page 908 option if not already present. If *prepend* is `1`, prepends *dir* to the *tagtemplatepath* path, removing any later occurrence of the directory.

If there is an *`Arbortext-path`*`\custom\tagtemplates` subdirectory at startup, the `custom\tagtemplates` path is automatically prepended to the tag template path.

You can 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.

Putting your tag template files in the `custom\tagtemplates` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

application_name

`application_name` (`[`*`untranslated`*`]`)

The `application_name` function returns the name of the application which is currently running, for instance Arbortext Editor, Arbortext Architect, Arbortext Publishing Engine Interactive or Arbortext Editor with Arbortext Styler. Your custom application might use this function to compare its result with a set of possible values.

If the optional *untranslated* parameter is present and not zero, the name is returned in English. If no parameter is given or if *untranslated* is specified as zero, this function returns the value of the built-in variable *$main::progname* on page 78. This value could be a translated version of the name (such as Arbortext Editor con Arbortext Styler for the `es` Spanish locale).

apply

`layout::apply` (*`application`*`[, `*`flags`*`[, `*`doc`*`]]`)

This function uses the current print FOSI to add page layout structure to the document and then calls an application specific function. The markup consists of singletons with the `atipl` namespace prefix, for example `atipl:startpage`.

The *application* parameter is the name of a callable function that implements some application specific behavior. Most commonly it will specify a set of generic attributes of the `atipl` markup to achieve a set of formatting effects. The application function must accept a single parameter: the document to modify, and return a status of `0` for failure or `1` for success. A sample line numbering application is located in the `samples\linenumbering` folder of your installation directory.

The optional *flags* parameter is a bit mask used to control the markup inserted. If not specified, then all possible markup is inserted. The bits control the type of structured markup to add to the document. These bits are:

- `0x1` — *line*
- `0x2` — *entry*
- `0x4` — *row*
- `0x8` — *float*
- `0x10` — *column*
- `0x20` — *span*
- `0x40` — *page*

The following bits are used to control the behavior of the function:

- `0x1000` — *generate diagnostics*
- `0x2000` — *no atipl markup for lines that contain only generated text*

The optional *doc* parameter specifies the document to modify. The current document is assumed if this parameter is not specified.

If the operation fails the function returns `0`. If the operation succeeds then the function returns `1`.

For more detailed information on the page layout elements and their attributes, see http://www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

apply_profile_group

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_name* — The apply profile group for which the function returns the number of profile classes.
- *arr* — Associative array of the values of the profile classes included in the specified apply profile group.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

If more than one value exists for a particular profile class, the different values will be separated using the value separator. For example:

```
arr[alias1] = value
arr[alias2] = value1;value2
```

apply_profile_group_allowed

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*.

If the group is not allowed, `apply_profile_group_allowed` returns `0` and fills the array *arr[]* with the alias or aliases of the invalid profiles that cause the group to not be allowed. The array is associative. It has the alias name as the index and the profilenode for the alias as the value. For example,

```
arr[ "Security"] = (1, 1, 1)
```

apply_profile_group_value_nodes

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

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

- *apply_profile_group_name* — The apply profile group for which the function returns the number of value profilenodes of the profile classes.
- *arr* — Array of the value profilenodes of all the profile classes that are included in the specified apply profile group.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

apply_profile_groups

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

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

- *arr* — Array of the names of all the apply profile groups specified in the profile configuration file.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

attr_alias

attr_alias (*attr*[, *element*[, *doc*]])

This function returns the alias of the attribute *attr*, where *attr* is the real name of the attribute as defined in the DTD, or the alias. Specify *element* if you want this function to return an element-specific attribute alias (*element* can be the real name or an alias). If *element* is omitted, this function returns the alias of a global attribute. A null string is returned if *attr* does not have an alias.

Note

If *element* is specified and there is no element-specific alias for *attr*, this function will return a null string even if a global attribute exists for *attr*.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

attr_description

attr_description (*attr*[, *element*[, *doc*]])

This function returns the description for *attr*. Specify *element* if you want this function to return an element-specific attribute description. If *element* is omitted, this function returns the description of a global attribute. A null string is returned if *attr* does not have a description.

Note

If *element* is specified and there is no element-specific attribute description for *attr*, this function will return a null string even if a description exists for a global attribute *attr*.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

attr_real_name

attr_real_name (*attr*[, *element*[, *doc*]])

This function returns the real name as specified in the DTD of the specified *attr*, where *attr* is the real name or alias of the attribute. Specify *element* if you want this function to return the real name of an element-specific attribute (*element* can be a real name or an alias). If *element* is omitted, this function returns the real name of a global attribute.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

attr_value_alias

attr_value_alias (*value*[, *attr*[, *element*[, *doc*]]])

This function returns the alias of the specified attribute *value*, where *value* is the real name of the attribute value as defined in the DTD, or its alias. Specify *attr* and *element* if you want this function to return the alias of an element-specific attribute value (*element* and *attr* can be real names or aliases). If both *attr* and *element* are omitted, this function returns the alias of a global attribute value. A null string is returned if *value* does not have an alias.

Note

If *element* is specified and there is no element-specific alias for *value*, this function will return a null string even if a global attribute value exists for *value*.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

attr_value_real_name

attr_value_real_name (*value*[, *attr*[, *element*[, *doc*]]])

This function returns the real name as specified in the DTD of the specified attribute *value*, where *value* is the real name or alias of the attribute value. Specify *attr* and *element* if you want this function to return the real name of an element-specific attribute value (*element* and *attr* can be real names or aliases). If both *attr* and *element* are omitted, this function returns the real name of a global attribute value.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

base_tag_name

base_tag_name (*tagname*[, *doc*])

This function returns the base element name for the user-defined tag named *tagname*. Returns null if *tagname* is not a valid element name. If the tag, *tagname*, is a base element, and not a user-defined tag, returns *tagname*.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

basename

basename (*path*[, *sfx*])

This function returns the base name of the file specified by *path*, removing any leading directory components and the suffix specified by *sfx* if present. If *sfx* is `.*`, then all characters following a trailing `.` are removed, including the period.

blength

blength(*str*)

Returns the number of bytes in the string specified by *str*. This contrasts with the `length` function which returns the number of characters. For example, `blength("abc")` returns `6` whereas `length("abc")` returns `3`.

buffer_clipboard_contents

buffer_clipboard_contents(*buf*[, *format*])

This function copies the contents of a specified format from the system clipboard to the scalar variable *buf*. If *format* is omitted or empty, then the standard clipboard text format is copied and the function returns the number of characters copied into *buf*. If *format* is specified and matches an application registered clipboard format (for example, `rich text format`), then that format is copied and the function returns the number of bytes copied into *buf*.

If the `set selectionsvc` option is off, this function always returns `0`.

Following is an example of a paste callback that pastes Rich Text Format content, instead of normal text content, when RTF is available:

```
global PASTE_RTF_BUFFER_NAME = 'rtfpaste';
function cb_rtf_paste(doc, buff, op) {
 if (op != 2) {
 return 0;
}
 # Check if there is RTF available on the system clipboard
 if (! buffer_is_clipboard("rich text format")) {
 return 0;
}
 local text;
 local len = buffer_clipboard_contents(text, "rich text format");
 if (len == 0) {
 return 0;
}
 # There is, so create a named paste buffer
 local second_doc = buffer_create(PASTE_RTF_BUFFER_NAME, public_id(doc));
 current_doc(second_doc);
 # and insert the RTF
 insert(mbstoucs(text, len));
 current_doc(doc);
 # remember original paste buffer
 local org_buffer = option('paste');
 # paste from our named paste buffer
 set paste = $PASTE_RTF_BUFFER_NAME;
 paste;
 # reset paste buffer
 if ('' == org_buffer) {
 set paste = default;
} else {
 set paste = $org_buffer;
}
 # delete the special buffer
 delete_buffer $PASTE_RTF_BUFFER_NAME;
 return 1;
}
```

buffer_clipboard_formats

buffer_clipboard_formats(*arr*)

This function fills the array *arr* with the names of the application registered clipboard formats currently available on the system clipboard. The function returns the number of names loaded into the array.

If the `set selectionsvc` option is off, this function always returns `0`.

buffer_create

buffer_create (*name*[, *pubid*[, *subtype*]])

This function creates a new named paste buffer *name* and returns a document identifier that may be used on subsequent calls to functions that operate on document trees. The *pubid* and *subtype* arguments are used to specify the document type for the paste buffer. See the `doc_open` function on page 325 for details. If omitted, the document type of the current document is used.

Note

If the paste buffer *name* exceeds the number of characters the operating system file naming convention accepts, the name is truncated if the paste buffer is saved as a file. You may later have difficulty locating the file when the buffer is loaded.

buffer_doc

buffer_doc (*name*)

This function returns the document identifier of the paste buffer named by *name*. `buffer_doc` returns `-1` for the current paste buffer. `buffer_doc` also returns `-1` if *name* is `default`, or if the named buffer does not exist.

buffer_empty

buffer_empty (*[name]*)

This function can be used to tell whether a local buffer contains data. If *name* is omitted, the default paste buffer is used.

If this function is called from a paste callback function, it will return 0 since the paste buffer always has contents at that point.

buffer_is_clipboard

ret = **buffer_is_clipboard**([*format*])

This function returns 1 (True) if the system owns the specified format on the clipboard. If *format* is omitted or empty, then the standard clipboard text format is assumed and the function returns 0 (False) if either Arbortext Editor's default paste buffer is not empty (meaning Arbortext Editor owns the clipboard) or the system clipboard is empty. If the set selectionsvc option is off, this function always returns 0.

This function will return 0 if called from a paste callback function when *format* is omitted or empty, since at that point Arbortext Editor owns the clipboard.

buffer_is_table

ret = **buffer_is_table**(*name*, *$gridId*[, *external*])

This function returns 1 (true) if the content of the paste buffer *name* represents a single table for the current document. If *name* is the empty string, the default paste buffer is assumed.

If buffer_is_table returns 1, *$gridId* is set to the TOId (table object identifier) for the grid.

If *external* is omitted or 0, the system clipboard is not checked. If *external* is not 0, the content of the system clipboard is checked to see if it represents the markup of a single table for the current document.

bulleted_list_block_tag_name

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

This function returns the name of the primary bulleted list block tag specified in the document type configuration file (.dcf) for the document type associated with *doc*. This tag is inserted when users select the bulleted list button on the Application toolbar.

If a tag name is returned, bulleted_list_block_tag_name function returns *arr* with an attribute name and attribute value, if they were specified for the tag in the .dcf file.

The function returns the null string if there isn't a bulleted list block tag designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or 0, the current document is used.

For example,

```
local tag, arr[];
local attr, val;
tag = bulleted_list_block_tag_name(arr);
```

```
attr = arr['attribute'];
val = arr['attribute_value'];
```

bulleted_list_block_tag_name_ns

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

This function returns the namespace URI prefixed to the primary bulleted list block tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted when users select the bulleted list button on the Application toolbar.

If a URI is returned, `bulleted_list_block_tag_name_ns` function returns *arr*.

The function returns the null string if there isn't a namespace prefix for the tag, if no bulleted list block tag is designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

bulleted_list_item_tag_name

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

This function returns the name of the primary bulleted list item tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted, along with the bulleted list block tag, when users select the bulleted list button on the Application toolbar.

If a tag name is returned, `bulleted_list_item_tag_name` function returns *arr* with an attribute name and attribute value, if they were specified for the tag in the `.dcf` file.

The function returns the null string if there isn't a bulleted list item tag designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

For example,

```
local tag, arr[];
local attr, val;
tag = bulleted_list_item_tag_name(arr);
attr = arr['attribute'];
val = arr['attribute_value'];
```

bulleted_list_item_tag_name_ns

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

This function returns the namespace URI prefixed to the primary bulleted list item tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted, along with the bulleted list block tag, when users select the bulleted list button on the Application toolbar.

If a URI is returned, `bulleted_list_item_tag_name_ns` function returns *arr*.

The function returns the null string if there isn't a isn't a namespace prefix for the tag, if no bulleted list item tag is designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

caller

`caller``([`*`level`*`])`

This function returns the name of the function that called this routine. If the argument *level* is given, it specifies the number of call frames to go back before the current one to obtain the information. Use either `caller()` or `caller(0)` to return the name of the current function.

caller_file

`caller_file` `([`*`level`*`])`

This function returns the file name containing the statement that called this routine. If the argument *level* is given, it specifies the number of call frames to go back before the current function to obtain the file name of the calling function. For example, `caller_file(1)` returns the file name of the call to the active function, whose name is returned by `caller(0)`.

caller_line

`caller_line` `([`*`level`*`])`

This function returns the line number of the statement that called this routine. If the argument *level* is given, it specifies the number of call frames to go back before the current function to obtain the line number of the calling function. For example, `caller_line(1)` returns the line number of the call to the active function, whose name is returned by `caller(0)`.

The `caller`, `caller_line` and `caller_file` functions can be used to generate a function trace back of the active ACL stack frame.

Example

```
function show_callers()
{
 local n = 0;
```

```
 while (caller_line(n)) {
 local c = caller(n);
 if (c == "") {
 c = "*top-level*";
}
 eval "Caller[" . n . "] is " . c output=>*;
 eval " at line", caller_line(n), "file", caller_file(n) output=>*
 n++;
}
}
```

caret_at

caret_at ([*doc*])

This function specifies the type of object preceding the cursor by returning one of the following strings: `text`, `tag`, `equation`, or, in the case of an empty file, `null`. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

caret_in_selection

caret_in_selection ([*doc*])

This function returns `1` (true) if the caret is within the current selection of the document specified by *doc*, or the current document if *doc* is omitted. Returns `0` (false) if the caret is not within the selection, or if nothing is selected in the document identified by *doc*.

caret_show

caret_show ([*doc*])

If the caret is within a collapsed region, this function expands all parent elements so that the caret is visible. The function can be directed to operate on a specific document by supplying the optional *doc* parameter. If *doc* is omitted, then the current document is used. If the current view is a document map, the hierarchy will be expanded sufficient levels to display the element containing the text caret.

catalog_ids

catalog_ids (*kword*, *arr*[, *regexp*[, *path*]])

This function returns all identifiers found in the catalog files specified by the path *path* for the specified keyword *kword* and match the regular expression specified by *regexp*. The number of entries found is returned.

The function has the following parameters:

- *kword* — A string that is a keyword indicating the type of entries that should be matched.

  This parameter is required and is case insensitive. The following keywords are valid:

  - `PUBLIC` — Entries using public identifiers, such as `PUBLIC` and `DTDDECL` entries.
  - `NAMES` — Entries using names, such `ENTITY`, `DOCTYPE`, `LINKTYPE`, and `NOTATION` entries.
  - `SYSTEM` — Entries using system identifiers.
  - `URI` — Entries using universal resource identifiers
  - `DOCBURST` — Entries using bursting specification identifiers
- *arr* — The name of the numerical array that contains the list of identifiers.

  This is a required parameter.
- *regexp* — A string that is a regular expression for filtering the entries.

  This parameter is optional with `*` being the default.
- *path* — A string that is the catalog path to use.

  This parameter is optional with the value of `option("catalogpath")` as the default.

Examples

```
# Find the public identifiers for DITA DTDs in the current catalog path.
local A[], cnt
cnt = catalog_ids("PUBLIC", A, "-//.*//DTD DITA")
# Find the universal resource identifiers for DITA schema modules in the current catalo
cnt = catalog_ids("URI", A, ":dita:.*:xsd:")
```

catalog_public_ids

`catalog_public_ids` (*`regexp`*, *`arr`*[, *`path`*])

This function fills the array *arr* with all the catalog entries in the specified catalog *path* list that match the regular expression *regexp*. The *path* parameter is optional; if it is not specified, the default catalog path is searched.

This function returns the number of entries found.

Example

```
local dtds[];
catalog_public_ids("-//.*//DTD", dtds,
 main::ENV["APTCATPATH"]);
```

This example defines the `dtds` array, then fills the array with all the catalog entries that match the regular expression `-//.*//DTD`. This example searches the catalog path defined in the *APTCATPATH* environment variable.

catalog_resolve

`catalog_resolve` (*`keyword`*`[, `*`p1`*`[, `*`p2`*`[, `*`p3`*`[, `*`catalogpath`*`]]]])`

This function is a general catalog lookup routine. The *keyword* argument specifies the catalog entry to search. The value of the *p1*, *p2*, and *p3* parameters depends on the *keyword* specified. The following table describes the values for each of the valid keywords.

keyword	***p1***	***p2***	***p3***
`DOCTYPE`	name	pubid	sysid
`ENTITY`	name	pubid	sysid
`NOTATION`	name	pubid	sysid
`PUBLIC`	pubid	sysid	
`SYSTEM`	sysid		
`DTDDECL`	pubid		
`SGMLDECL`			
`FOSI`			
`URI`	uri		

pubid is the public identifier and *sysid* is the system identifier.

The *catalogpath* argument specifies a list of directories to search for the `catalog` file. If it is not specified, Arbortext Editor searches the path list specified in the `set catalogpath` option, which is initialized by the *APTCATPATH* environment variable.

uri is the namespace URI.

When specifying the *catalogpath* argument, use null for trailing unused parameters. For example, `catalog_resolve("SGMLDECL", "", "", "", catpath)`.

The `public_id_path(pubid, "", catpath)` function is equivalent to `catalog_resolve("PUBLIC", pubid, "", "", catpath)`.

The `dtd_decl_path(pubid, catpath)` function is equivalent to `catalog_resolve("DTDDECL", pubid, "", "", catpath)`.

The `entity_resolve(name, pubid, sysid, catpath)` function is equivalent to `catalog_resolve("ENTITY", name, pubid, sysid, catpath)`.

The `uri_resolve(uri, catpath)` function is equivalent to `catalog_resolve("URI", uri, "", "", catpath)`.

catch

`catch``(`*`expr`*`[, `*`noerr`*`])`

This function evaluates the expression *expr* and traps any run-time or parser errors resulting from evaluation of the expression. It also allows non-local exits from functions invoked by the expression using `throw`. If the optional argument *noerr* is specified and non-zero, then any error messages can be suppressed as with `eval`.

`catch` returns zero if the expression is evaluated without error. If there was an evaluation error, `catch` returns 1. If `throw` was executed during the evaluation of *expr*, `catch` returns the value of `throw`, which must be non-zero.

change_tracking_accept_change

`ret = `**`change_tracking_accept_change`**`(`*`oid`*`[, `*`flags`*`])`

This function attempts to accept a tracked change in the document and returns the status of the operation. The *oid* parameter specifies the location of the change markup. The *flags* parameter is a bit mask used to control the operation.

If *oid* is not valid change markup,then this function returns `-1`. If accepting the change would cause an in-context document to become out of context and *nocc* bit is not `1`, a confirmation is displayed to allow the user to cancel the acceptance. If the user cancels the command then this function returns `0`. In either case this function does not modify the document. Otherwise, the change is successfully accepted and this function returns `1`.

The flag bit is:

- 0x1 — *nocc*

Note

Accepting a change can be undone with the `undo` command.

change_tracking_accept_selection

`ret = `**`change_tracking_accept_selection`**`(`*`isValid`*`[, `*`doc`*`[, `*`oid`*`]])`

This function attempts to accept a tracked change in selected content in the document *doc*.

- *isValid* — A value specifying how the function handles the change.
  - If *isValid* is non-zero, `change_tracking_accept_selection` determines whether there is a selection that can be accepted. The function returns `1` if there is a current selection which is balanced and completely inside of a highlighted add or delete change tracking region. Otherwise, the function returns `0`.
  - If *isValid* is `0`, `change_tracking_accept_selection` determines if the selection is valid to accept, and then attempts to accept the selection. Accepting the selection may result in splitting the change tracking region into two regions. If the accept succeeds, the function returns `1`. Otherwise, the function returns `0`.
- *doc* — Optional. The document in which to accept changes. If *doc* is omitted, the current document is used.
- *oid* — Optional. If *isValid* is non-zero and *oid* is supplied, then *oid* is set to the change tracking markup that is available to be accepted.

change_tracking_find

`oid2 = change_tracking_find(oid1[, offset[, flags]])`

This function searches for change tracking markup, highlights it and returns the location in *oid2*. It returns the null oid if no change tracking markup is found.

The first parameter, *oid1*, is the starting location of the search. If null, the start of the current document is assumed, or the end if searching backward. The second parameter, *offset*, is the offset from *oid1* within the following text. The search does not include the *oid1* tag itself, even when the offset is zero.

The third parameter, *flags*, is a bit mask used to control the search. The possible flag bits are:

- 0x1 — *parent/child*
- 0x2 — *reverse*
- 0x4 — *no select*
- 0x8 — *no message*

By default, the search will move forward through the document and highlight any changes found. Wrapping is controlled by the `wrapscan` set option.

The *parent/child* bit, if set, causes the search to search the document or to recursively search children for change tracking markup.

The *reverse* bit reverses the direction of the search.

The *no select* bit is used to turn off the highlighting.

The *no message* bit is used so the function will not display a “not found” message. This does not affect the prompt for wrapping which is controlled by the `set wrapprompt` command.

change_tracking_info

`cnt` = **`change_tracking_info`**`(`*`oid`*`, `*`array`*`)`

This function populates the *array* parameter with information about the change tracking markup at *oid* and returns the number of elements in the associative array. If the *oid* is not change tracking markup, then the function returns `0`.

The following fields may appear in the associative array:

- `type` — The type of change markup (for example, “add”, “del”).
- `subtype` — Subtype if present (for example, “text”).
- `user` — ID of user who made the change (for example, “ref”) to be used with `change_tracking_user_properties`.
- `time` — Time of change (expressed in seconds since 1/1/70) to be used with the function `ctime` on page 254.
- `visible` — `1` if the current view setting shows this change. `0` otherwise.
- `unacceptable` — A description of why attempting to accept this change would fail.
- `unrejectable` — A description of why attempting to reject this change would fail.
- `nested` — A message explaining that acceptance or rejection of this change would remove another change.

change_tracking_reject_change

`ret` = **`change_tracking_reject_change`**`(`*`oid`*`[, `*`flags`*`])`

This function attempts to reject a tracked change in the document and returns the status of the operation. Rejecting the change is similar to undoing the change. The *oid* parameter specifies the location of the change markup. The *flags* parameter is a bit mask used to control the operation.

If *oid* is not valid change markup,then this function returns `-1`. If there is a calling sequence error then this function returns `-1`. If rejecting the change would cause an in-context document to become out of context and *nocc* bit is not `1`, a confirmation is displayed to allow the user to cancel the rejection. If the user

cancels, then this function returns `0`. In either case this function does not modify the document. Otherwise, the change is successfully rejected and this function returns `1`.

The flag bit is:

- 0x1 — *nocc*

Note

Rejecting a change can be undone with the `undo` command.

change_tracking_reject_selection

`ret = change_tracking_reject_selection(isValid[, doc[, oid]])`

This function attempts to reject a tracked change in selected content in the document *doc*.

- *isValid* — A value specifying how the function handles the change.
  - If *isValid* is non-zero, `change_tracking_reject_selection` determines whether there is a selection that can be rejected. The function returns `1` if there is a current selection which is balanced and completely inside of a highlighted add or delete change tracking region. Otherwise, the function returns `0`.
  - If *isValid* is `0`, `change_tracking_accept_selection` determines if the selection is valid to reject, and then attempts to reject the selection. Rejecting the selection may result in splitting the change tracking region into two regions. If the reject succeeds, the function returns `1`. Otherwise, the function returns `0`.
- *doc* — Optional. The document in which to reject changes. If *doc* is omitted, the current document is used.
- *oid* — Optional. If *isValid* is non-zero and *oid* is supplied, then *oid* is set to the change tracking markup that is available to be rejected.

change_tracking_user_list

`cnt = change_tracking_user_list(array[, doc])`

This function populates the indexed *array* with the unique ids of all users in the user list for the document *doc*. If *doc* is omitted, all users in the internal user table will appear. The function returns the number of entries in the array.

change_tracking_user_properties

`cnt = change_tracking_user_properties(user, array[, set])`

This function populates the associative *array* with the set of known properties for the user identified by the unique id *user*. If the optional *set* parameter is non-zero, the *user* is added to the internal table if needed, and the properties in the array are associated with that user in the internal user table.

The keys of the associative array of properties for a new user will include:

fullname — The user's full name.

Any key/value pair may be added.

The internal user table data for a given user is saved with any document that has change records created by that user at the time that the document is written out. A document's user table, including all user properties, is merged into the internal user table when that document is read in a future session.

char_entity_names

`char_entity_names (arr[, doc])`

The `char_entity_names` function fills the array *arr* with an alphabetical list of character entity names which are valid in the current document type, returning the number of entities.

The first name is stored at index 1. The *doc* argument specifies the identifier of the document tree to be queried. If omitted or `0`, the current document is used.

chop

`chop(varname[, len])`

This function removes the last *len* characters from the variable named by *varname*, which must be a scalar variable or an array element. The `chop` function returns the characters removed. If *len* is omitted, the default `1` is used, removing just the last character. This is useful for removing the trailing newline character returned by `getline`.

chr

`chr(n)`

This function returns a string of one character corresponding to the ASCII character number or Unicode character number *n*. This is the inverse of `ord`. For example, `chr(65)` is `A`, and `chr(ord('X')) == 'X'`.

cjk_locale

cjk_locale ()

This function returns the following non-zero values if Arbortext Editor is running on a Chinese, Japanese or Korean (CJK) system locale:

1	Japanese
2	Simplified Chinese
3	Traditional Chinese
4	Korean

It returns a 0 if Arbortext Editor is not running on a CJK system locale.

Execute the following command at the Arbortext Editor command line to determine the system locale:

```
eval cjk_locale()
```

clear

layout::clear ([*doc*])

This function removes the `atipl` namespace declaration and all page layout markup from a document and returns the status of the operation. The *doc* parameter specifies the document to modify. The current document is assumed if this parameter is not specified.

If the document contains page layout markup and the operation fails, the function returns `0` and the namespace declaration is retained. If the operation succeeds, then the function returns `1`.

For more detailed information on the page layout elements and their attributes, see www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

clear_stylesheet

clear_stylesheet (*path*)

This function clears the stylesheet indicated by *path* from the compiled stylesheet cache. Arbortext Editor stores XSL stylesheets in this cache after they've been compiled during a publishing process. By storing these stylesheets, Arbortext Editor avoids having to recompile them during subsequent publishing runs.

You can obtain a list of paths using the `list_stylesheets` function on page 404. If *path* is not specified or is an empty string, all stylesheets are cleared from the cache.

As an example, if you edited an XSL stylesheet that had previously been used in a publishing run, you would use this function to clear out the stylesheet cache. The updated stylesheet would then be used during the next publishing.

close

close(*fid*)

This function closes the file or channel associated with the identifier *fid*, which must be a return value of a previous call to `open`, `open_accept`, `open_connect`, or `open_listen`. `close` returns `0` if the file was closed successfully, or `-1` if the file wasn't closed successfully (such as trying to close an already closed file).

cmd_exists

cmd_exists (*cmdname*)

This function returns `1` (True) if the string specified by *cmdname* is a valid built-in command or a currently defined alias. For built-in commands, *cmdname* may specify an unambiguous initial substring or abbreviation. If the command or alias does not exist or is not valid, `0` is returned.

cmd_key

cmd_key(*alias*[, *keymap*])

This function returns the keyboard shortcut bound to the command alias specified by *alias* for the keyboard shortcut specified by *keymap*. The result is a string of the same form as the *keyname* argument for the map on page 669 command (for example, `Ctrl+Shift+A`) or key_cmd on page 398 function.

keymap is the same as the window argument to `map` and is a window class (for example, `edit`, `cmd`, `helpwin`), a window name as returned by the `window_name` function, or a user-defined keyboard shortcut identified by the prefix character `@` (for example, `cmd_key("FileSave", "@panel")`). If *keymap* is omitted, the keyboard shortcut associated with the current window is used.

The function returns a null string if *alias* is not bound to any keys in the specified keyboard shortcut .

color_chooser

color_chooser (*color*)

This function displays a color selection dialog box and returns the user's choice of color. The null string is returned if the user selects **Cancel**. *color* specifies the initial color to edit and is either a named color or an RGB specification preceded by `#`.

Example

```
clr = color_chooser("#ff0000");
myred = color_chooser("red");
```

The look of the Color Chooser dialog box varies across platforms.

color_rgb

color_rgb (*color*[, *background*])

This function returns the RGB number of the color specified in the *color* parameter. The *color* parameter can be either a named color an RGB specification preceded by `#`. When the *color* parameter is set to a named color and the optional *background* parameter is set to `0`, the foreground color of the named color is returned. If *background* parameter is not set to `0`, the background color of the named color is returned.

columnview_cell_focus

columnview_cell_focus([*window*[, *newcol*]])

This function enables you to change the focus from one Column view cell to a different cell.

The *window* parameter is the view for which you want to change the cell focus. If *window* is omitted or set to zero, the current window is used.

The *newcol* parameter is the column number of the cell that you want to assume the focus. Column zero is the **Outline** column. If a column number larger than the number of columns being displayed is given, the focus is moved to the last cell of the row. If *newcol* is omitted or set to a negative value, the cell focus is not changed.

The function returns the number of the column that currently has the focus. Column zero is the **Outline** column. A value of `-1` indicates that the view specified by *window* is not a Column view or is otherwise invalid.

columnview_is_defined

columnview_is_defined([*doc*])

This function indicates whether Column view is defined in the document type configuration file (`.dcf`) for the indicated document.

The *doc* parameter is the document for which you are performing the check. If *doc* is omitted or set to zero, the current document is used.

The function returns `1` (true) if Column view is defined in the `.dcf` file or `0` (false) if Column view is not defined. A Column view does not need to be currently displayed for this function to return true.

com_attach

`handle = **com_attach** (*progID*)`

Returns a handle to a COM object which can be used to invoke a method or access a property on that object. The object must be released when it is no longer needed using `com_release` on page 248.

The *progID* parameter is either the `ProgID` or the `CLSID` of the COM object to attach to. A `ProgID` is the name registered for the COM object, such as `Excel.Application`. A `CLSID` is a character string starting with `{` represents the GUID that uniquely defines the object to attach to. If the COM object is not running, it will be started either as a separate process or by loading a DLL into the Arbortext Editor process as appropriate. If the COM object doesn't exist or can't be invoked, the function returns a zero (0).

Example

```
wordh = com_attach("Word.Application")
```

The above code returns a handle that could be used to invoke Microsoft Word.

com_call

`result = **com_call** (*handle*, *intname*[, *arg1*[, *argn*]])`

Invokes a method in a COM object previously attached to using `com_attach` on page 246 or one returned by a method called previously.

The *handle* parameter is a handle returned by a call to `com_attach` on page 246 or returned by a previous call to a COM method or property.

The *intname* parameter is a character string that gives the name of the method to be invoked.

All other parameters (if any) are passed to the method named by *intname* in the object specified by the object's *handle*. Each parameter will be converted to the type expected by the interface being invoked.

The return value will be converted to an appropriate ACL type before being used. If it is a COM interface, it will be converted into a handle to the interface which can be used in subsequent calls to `com_call`. The handle must be released using `com_release` on page 248 when it is no longer needed.

Example

```
result = com_call(wordh, "Quit", -1)
```

invokes the `Quit` method in the COM object represented by *wordh*. If this is the handle returned by the example in com_attach on page 246, this will cause Microsoft Word to quit while saving open documents. The other two optional parameters to the `Quit` method are omitted.

Note

In COM interfaces, “True” is represented by minus one (-1) and “False” by zero (0).

com_prop_get

`result = com_prop_get(handle, prop[, arg1[, argn]])`

Retrieves the value of a property in a COM object previously attached to using com_attach on page 246 or one returned by a method called previously.

The *handle* parameter is an object handle returned by a call to `com_attach` or returned by a previous call to a COM method or property.

The *prop* parameter is a character string that gives the name of the property to be obtained.

All other parameters (if any) are parameters for the property. If a property requires an extra parameter, it is most likely an index for an indexed property. Each parameter will be converted to the type expected by the interface being invoked.

The return value will be converted to an appropriate ACL type before being used. If it is a COM interface, it will be converted into a handle to the interface which can be used in subsequent calls to com_call on page 246. The handle must be released using com_release on page 248 when no longer needed.

Example

```
docsh = com_prop_get(wordh, "Documents")
```

The above example gets a handle to the Documents collection assuming that *wordh* is the handle returned in the example for `com_attach`. This handle could then be used to open a document, and return a handle for it, using a call like the following:

```
doch = com_call(docsh, 'C:\Temp\Sample.DOC', 0, -1)
```

com_prop_put

`result = com_prop_put(handle, prop, newval[, arg1[, argn]])`

Sets the value of a property in a COM object previously attached to using com_attach on page 246 or one returned by a method called previously.

The *handle* parameter is a handle returned by a call to com_attach or returned by a previous call to a COM method or property.

The *prop* parameter is a character string that gives the name of the property to be set.

The *newval* parameter is the new value for the property.

All other parameters (if any) are parameters for the property. If a property requires an extra parameter, it is most likely an index for an indexed property. Each parameter will be converted to the type expected by the interface being invoked.

Setting a property doesn't normally return a value, but if it does, it will be converted to an appropriate ACL type before being used. If it is a COM interface, it will be converted into a handle to the interface which can be used in subsequent calls to com_call on page 246. The handle must be released using com_release when no longer needed.

Example

```
com_prop_put(doch, "DefaultTabStop", "36.5")
```

The previous example sets the default tab stop to 36.5 points. Note that when a parameter is a floating point number, you specify it as a string since ACL doesn't have full support for floating point numbers.

com_release

`result = com_release (handle)`

Releases a handle to a COM object which was previously obtained with com_attach on page 246 or as the result of a call to com_call on page 246 or com_prop_get on page 247. It is very important to release all COM handles when they are no longer needed since the COM server they are attached to will not exit properly if they are not released. Also, you should release all COM handles before you exit from Arbortext Editor.

The return value from com_release is 1 if it succeeded and 0 if it failed.

Example

```
com_release(wordh)
```

The example above releases the handle that was obtained in the example to com_call.

compare_files

compare_files (*previous_file*, *current_file*, *result_file*)

This function performs a comparison of two document files. The two files being compared are specified as the first two arguments. The third argument names the comparison results file that can be opened, displayed, and printed in Arbortext Editor.

The *previous_file* argument is the file name for one of the documents you want to compare. The *current_file* argument is the file name for the document that will be compared to *previous_file*.

The *result_file* argument is the file name you want to give to the comparison results document.

The return value from `compare_files` is 1 if it succeeded and there are differences in the two files. The return value is 0 if there are no differences in the files. The return value is -1 if it failed (for example, if an input file doesn't exist). If the return value is less than 1, then a results document isn't created and an error message is stored in the variable *main::ERROR*.

composer_log

composer_log (*destination*, *severity*, *verbosity*)

The `composer_log` function sets the preferences for the Event Log file. Use this log file to troubleshoot publishing problems.

destination specifies the output destination for publishing events:

Setting	Description
`0`	Outputs to the Java Console.
`1`	Outputs to the Event Log window.
string	Outputs a text file to the specified path.

severity specifies the minimum severity of output events to generate:

Setting	Description
$eventlog::SEVERITY_INFO	Includes all diagnostic events, including errors and warnings.
$eventlog::SEVERITY_WARNING	Includes errors and warnings, but no informational events. This is the default setting.
$eventlog::SEVERITY_ERROR	Includes errors only.

verbosity indicates the amount of contextual information to generate in the log:

Setting	Description
0	No contextual information.
1	Exception message.
2	Exception message and traceback. This is the default setting.

For example,

```
# Trace all possible publishing events to a file.
composer_log("c:\\temp\\event.log", $eventlog::SEVERITY_INFO, 2)
# Trace warnings and errors (but not informational events) to the
# java console and don't include traceback information.
composer_log(0, $eventlog::SEVERITY_WARNING, 1);
# Trace only errors (but not warnings and informational events)
# to the event log.
composer_log(1, $eventlog::SEVERITY_ERROR);
```

content_model

content_model (*tagname*[, *doc*[, *index*]])

This function returns the canonical form of the content model for the element specified by the expression *tagname*.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name. However, the function will return an element's real name, not its alias.

For example, a model declared to be

```
(a, (%ref;)+, d)
```

where the REF parameter entity is declared as

```
b|c
```

will be returned as

```
(a,(b|c)+,d)
```

`content_model` returns the null string if *tagname* is not an element declared in the DTD. The tagname may start with the end tag open character / (ETAGO).

The optional *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Schemas can have no, one, or more than one content model. The optional *index* parameter lets you control the content model information returned.

- Setting *index* to `-1` — (The default.) Returns the global content model of the *tagname* in *doc*. If the element has no global content model, an empty string is returned.
- Setting *index* to a value `>=0` — Returns a global or local content model. However, the order of these models is undefined. You can enumerate through all content models by using `0`, `1`, `2`, `...` until an empty string is returned (the signal to stop). (For a DTD, `0` and `-1` always return the one and only content model and >=1 always return an empty string.)

context_full_paths

`context_full_paths` (*`arr`*, *`tag`*`[,` *`depth`*`[,` *`maxpaths`*`[,` *`oid`*`[,` *`pos`*`]]]])

Note

The `context_full_paths` function replaces the `context_paths` function on page 252, which is being deprecated.

This function fills the array *arr* with a list of possible context paths to make the target element *tag* in context at the point in the document given by *oid, pos*. If one of the paths returned is an empty string, *tag* can be inserted at the specified position without any elements being added.

Note

If you have applied an alias map to the document, *tag* can be an alias or a real name. However, this function will return an element's real name, not its alias.

If *oid* is omitted, the cursor position in the current document is used.

Depth specifies the maximum tag nesting depth of the paths returned. If *depth* is omitted, a depth of 5 is used.

Maxpaths specifies the maximum number of paths at each depth to return. It defaults to 50. If *depth* is 5, and *maxpaths* is 50, as many as 250 total paths could be returned. If more paths than *maxpaths* exist at a given *depth*, only the first *maxpaths* paths are returned, with no indication that more paths exist.

`context_full_paths` returns the number of paths stored in the array. Each path returned is expressed as a context string as formatted by the `context_string` on page 253 function. The array is ordered by path depth, with the shallowest paths first.

This function adds required sibling elements to the paths returned. For example, if the content model for chapter is (title, para*) and the cursor is positioned just inside a `chapter` tag, then `context_full_paths(arr,"para")` will return `"title()"`. On the other hand, if the content model for chapter is (title*, para*), where the title is not required, then `context_full_paths` will return an empty string, because `para` is valid within `chapter` without any elements being added.

The siblings that are included do not add to the depth. For example, the following paths both have a depth of two:

```
"abstract("
"title()subtitle()abstract("
```

context_paths

context_paths (*arr*, *tag*[, *depth*[, *oid*[, *pos*]]])

> **Note**
>
> The `context_paths` function is being deprecated. When writing new applications, use `context_full_paths` on page 251 instead.

This function fills the array *arr* with a list of possible context paths of no more than *depth* tags to make the target element *tag* in context at the point in the document given by *oid*, *pos*. If *oid* is omitted, the cursor position in the current document is used. If *depth* is omitted, a depth of 5 is used.

> **Note**
>
> If you have applied an alias map to the document, *tag* can be an alias or a real name. However, this function will return an element's real name, not its alias.

`context_paths` returns the number of paths stored in the array. Each path returned is expressed as a context string as formatted by the `context_string` on page 253 function. The array is ordered by path depth, with the shallowest paths first.

For compatibility with previous versions, each path returned starts with the enclosing tag at the cursor position or at the *oid pos* passed in. For example, if the cursor is just inside a `chapter` tag, then `context_paths(arr,"para")` will return an array of paths that start with "chapter(". If that is not the desired result, use the `context_full_paths` function when writing new applications.

The `context_paths` function does not add required sibling elements to the paths returned. For example, if the content model for chapter is *(title,para*)* and the cursor is just inside a `chapter` tag, then `context_paths(arr,"para")` will return `"chapter("` rather than `"chapter(title()"`. If you want required siblings to be added, use the `context_full_paths` function.

Note

The `context_paths` function returns an empty array if the cursor is positioned at the beginning of the document.

context_string

`context_string` `([`*`doc`*`])`

This function returns a string describing the context of the cursor in the current window. This string consists of a list of element names and parentheses, such as: `doc(body(chapter(title()para0(title()para(`

The left parenthesis following an element name represents a start tag, and the right parenthesis represents the end tag for the corresponding unmatched start tag. If the cursor is before the opening start tag or if context checking is not relevant for the current document, a null string will be returned.

Note

Even if you have applied an alias map to the document, this function will return an element's real name, not its alias.

The *doc* argument specifies the identifier of the document tree to be queried. If omitted or `0`, the current document is used.

count

`count``(`*`arr`*`)`

This function returns the number of elements in array *arr*.

create_copypaste_map

`create_copypaste_map` `(`*`source`*`,` *`destination_path`*`[,` *`doc`*`])`

This function creates a basic Arbortext Import/Export map template for a specified document source format and an XML document type. You can then use the Arbortext Import/Export **MapTemplate Editor** to customize that map for converting content of the given format to the XML markup for the document type. This map is used to convert source formats stored on the Microsoft Windows clipboard when copying and pasting text from other applications to Arbortext Editor.

The *source* parameter designates the document source format for which you want to generate the map template. The following values are supported:

- `rtf` — Rich Text Format (RTF), the document format supported by several Microsoft applications including Microsoft Word
- `htm` — HTML markup
- `mif` — Maker Interchange Format (MIF), the document format supported by Adobe FrameMaker
- `txt` — Unicode and 8-bit ANSI text

The *destination_path* parameter is the full path for the generated map template. To be recognized as a customized map for copy and paste conversion, the file must be named according to the following convention: *`source-doctypename`*`.std`, where *source_type* is the value of the *source* parameter and *doctypename* is the name of the specified document's document type. For example: `rtf-axdocbook.std`

The optional *doc* parameter designates the document for which you want to generate a map. If you do not supply a *doc* value, the current document is used.

The function returns `0` on success or an error code on failure. The following error code values are currently supported:

- `1` — Could not write to map template file
- `2` — Invalid *source* parameter value
- `3` — Invalid *doc* ID
- `4` — Invalid document type

ctime

`ctime``(`*`time`*`[, `*`gmt`*`])`

This function converts the value *time*, as returned by `time` or `file_mtime`, into a string of the form produced by `time_date`. If the optional argument *gmt* is specified and non-zero, the time is returned in *gmt*. Otherwise, the time is given in the local time zone.

current_doc

current_doc ([*doc*])

This function returns the document identifier of the current document tree. If *doc* is given, this function sets the current document to the specified identifier and returns the previous value. Subsequent commands in the current execution context (either function or alias) will operate on this document tree. The *doc* argument must be a return value from a previous call to `doc_open`, `window_doc`, or `oid_doc`.

current_event

current_event ([*detail*])

This function returns information about the event that caused the command script containing this function call to be executed. The return value is one of the strings below. If the *detail* parameter is specified, additional details will be provided about the event.

- `key` — a key is pressed. *detail* is the name of the key, for example, `Ctrl+Shift+F3`.
- `toolbar` — a toolbar button is pressed. *detail* is the name of the toolbar button, for example, `Toolbar_Save`.
- `mouse` — a mouse button is pressed. *detail* is the name of the button, for example, `Ctrl+M1`.
- `menu` — a menu is selected. *detail* is the name of the menu item, for example, `Save As...`.
- `scroll` — a scrollbar event is active. *detail* is null.
- `other` — some other event is active. *detail* is null. Scripts are unlikely to get this return value.
- `none` — no event is active. For example, the script is run outside of Windows mode. *detail* is null.

This function is useful to make a command behave differently depending on whether it is mapped to a key or menu.

current_tag_attr_value

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

This function returns the current, or local, value of the attribute named *attr* for the element markup preceding the cursor. If the markup is an end tag, then attribute values for the start tag are used. The *doc* argument specifies the identifier of the document tree to be queried. If omitted or `0`, the current document is used.

Note

If you have applied an alias map to the document, *attr* can be either an alias or a real name. However, this function will return the attribute value's real name, not its alias.

current_tag_name

`current_tag_name` `([`*`doc`*`])`

In the Edit view, this function returns the name of the tag to the left of the cursor. In the Document Map view, this function returns the name of the tag to the right of the cursor if the cursor is at the start of a line (that is, between the element icon and the element name); you can control this behavior with the `set docmapcurrenttag=off` option.

Note

Even if you have applied an alias map to the document, this function will return a tag's real name, not its alias.

This is similar to the *tagname* variable except the current window is used. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

current_window

`current_window` `([`*`window`*`[, `*`doc`*`]])`

This function returns the window identifier of the active window associated with the document specified by *doc*, or the current document if *doc* is omitted or `0`.

If *window* is specified, this function sets the active window for the document to this window, and makes *doc* the current document. The keyboard focus remains in the same pane as the previous frame.

If no parameters are specified, `current_window` returns the window which has focus at the time the function is called. Specifying a parameter changes keyboard focus to the specified window for the frame containing that window. For example, you can determine which pane in a split window will get keyboard focus the next time the frame gets focus.

cut_valid

cut_valid ([*doc*[, *ignoreROFileEnt*]])

This function returns `1` (True) if the selected region can be cut without causing context errors. It returns 0 if the region is not balanced, is a protected region, or if context errors would result if the region was deleted. The *doc* argument specifies the identifier of the document owning the selection. If omitted or `0`, the current document is used.

If the *ignoreROFileEnt* argument is supplied and is not `0`, any unlocked file entities or XML inclusions that contain the selection will be ignored with respect to determining if the selection is protected.

Note, this function does not consider fix ups (such as adding required tags) when determining whether the cut will succeed.

dcf_option

dcf_option (*attrname*[, *doc*])

This function returns the value from attributes on the `Options` element in the document type configuration file (`.dcf`). The *attrname* argument is the name of an attribute on the `Options` element (case is significant). The *doc* argument is an optional document identifier. The `.dcf` for the document type associated with *doc* is used for this function. If *doc* is not provided or set to zero, the current document is used.

The function returns the value of the given attribute or a null string. A null string is returned in the following cases:

- The attribute has no assigned or default value.
- The given attribute name is not a valid `.dcf` `Options` element attribute name.
- The given document is not a valid document identifier.

dcf_validate

dcf_validate ([*doc*])

This function reads the document type configuration file (`.dcf`) for the document type associated with *doc*, and checks that its contents are valid. It does not do a context check.

If *doc* is a `.dcf` file, then *doc* is verified. Otherwise, the `.dcf` file for the document type for *doc* is used. If *doc* is omitted or `0`, the current document is used.

This function returns:

Value	Description
`-1`	Couldn't load the corresponding document type (needed to verify element and attribute names).
`0`	Detected errors or warnings, which are displayed in a message window.
`1`	Didn't detect any errors or warnings.

Note

Arbortext Editor only detects errors, not warnings, when a `.dcf` file is loaded.

Following is a list of errors that the `dcf_validate` function detects:

- Graphic definition missing both entity and file name attributes.
- Graphic definition with scaleToFit attribute set, but with either reproDepth or reproWidth missing.
- Link definition missing both uri and idref attributes.
- Target definition missing id attribute on an element without an attribute named "id".
- ElementDisplay entry contains pair="end" for a singleton entry.
- Icon entry contains invalid character set or character number.

Following is a list of warnings that the `dcf_validate` function detects:

- Element name is not found in the target DTD.
- Attribute name is not found on element in the target DTD.

Note

This means that Arbortext Editor only verifies the attribute when it is paired with an element attribute. The attribute values in the `AttributeOption` element are not checked because they apply globally to the document type.

- Element name is duplicated in a InsertAroundToFix list.
- Element name is duplicated in an element's Substitutions list.
- Element name is duplicated in a Smart Insert Category list.
- Substitution list or InsertAuto contents for an element is empty.
- InsertAutoWithin section contains multiple InsertAutoCaret or InsertAutoSelection elements.
- InsertAutoWithin section for a singleton element contains an InsertAutoSelection element.
- ElementDisplay entry refers to an icon that is not declared.
- Smart Insert list for a category is empty.
- Allowed value is duplicated in a profile class.
- Profile class has an empty allowed value.
- InsertAuto content found for a table element (except when in a cell or caption).

Note

InsertAuto content is inserted unless markup is inserted by the Table Editor.

- InsertAutoCaret or InsertAutoSelection found in InsertAuto contents for a table cell element.

Note

InsertAutoCaret or InsertAutoSelection are ignored when a cell is added to a table using the Table Editor.

dcfmodel_element_list

dcfmodel_element_list(*arr*, *type*[, *doc*[, *dcfOnly*]])

This function returns the array *arr* with a list of elements designated as a given *type* in the document type configuration file (`.dcf`) for the document type associated with *doc*. The function also returns the number of entries in array *arr*. If *doc* is omitted or `0`, the current document is used.

If the document is using a Arbortext Styler stylesheet (`.style`), data on graphic, link, and link target types is obtained from the `.style` file rather than the `.dcf` file. If the document is not using a Arbortext Styler stylesheet, or if the *dcfOnly* parameter is set to `1`, all data is retrieved from the `.dcf` file.

Following are the valid *type* values that are sorted. (The primary element is returned first.)

- bold
- bulleted_list
- graphic
- italic
- link
- numbered_list
- paragraph
- target
- underline
- smallcaps
- subscript
- superscript

Note

The `SmallCaps`, `Subscript`, and `Superscript` text styles are not supported in this release.

Following are the valid *type* values that are not sorted:

- division
- division_title
- list_block

- list_item
- webcompose_boundary

declare_char_entity

declare_char_entity (*name*[, *text*[, *doc*[, *inEntity*]]])

This function allows you to declare a character entity with a specific replacement value.

name is the name of an existing file entity. A leading `&` is optional.

text is the replacement value for *name*, and is usually the name of a special character inside brackets. The *text* argument in aliases must be enclosed by quotes if brackets are used.

doc is the document identifier. By default, the current document is assumed.

inEntity signifies whether the declaration is within an entity. A non-zero value means the declaration is within an entity.

declare_file_entity

declare_file_entity (*name*[, *sysid*[, *pubid*[, *type*[, *doc*[, *confirmDtdOverride*[, *inEntity*]]]]]])

This function allows you to declare a file entity with a specific name.

name is the name of an existing file entity. A leading `&` is optional.

sysid is the new system identifier value (if there is one).

pubid is the new public identifier value (if there is one).

type is either empty, `normal`, or `SUBDOC` to declare the entity as a SUBDOC entity.

doc is the document identifier. By default the current document is assumed.

confirmDtdOverride prompts the user whether to override the DTD if the entity is already declared in the DTD. A non-zero value will prompt the user.

inEntity signifies whether the declaration is within an entity. A non-zero value means the declaration is within an entity.

declare_filep_entity

declare_filep_entity (*name*[, *sysid*[, *pubid*[, *doc*[, *inEntity*]]]])

This function adds a declaration for a file parameter entity (that is, an external file parameter entity) to the document's internal subset.

name is the name of the entity. A leading percent sign `%` is optional. If the entity hasn't already been declared, an error message is displayed.

sysid is the entity's system identifier.

pubid is the entity's public identifier.

doc is the document identifier. By default, the current document is assumed.

inEntity signifies whether the declaration is within an entity. A non-zero value means the declaration is within an entity.

For example, after the function call:

```
declare_filep_entity("myent", "/usr/myent")
```

the document's internal subset declaration will contain this:

```
<!ENTITY % myent SYSTEM "/usr/myent">
```

The declaration is added to the beginning of the subset.

Note

To ensure the declarations in the file parameter entity are added to the document's declaration, save the document, then choose **File ► Revert to Saved**.

declare_graphic_entity

declare_graphic_entity(*name*[, *notn*[, *sysid*[, *pubid*[, *doc*[, *confirmDtdOverride*[, *inEntity*]]]]]])

This function allows you to declare a graphic entity with a specific name and a specific notation.

name is the name of an existing graphic file entity. A leading `&` is optional.

notn is the name of the graphic entity's notation. A notation can associate a graphics application with a specific type of graphic file.

sysid is the new system identifier value (if there is one).

pubid is the new public identifier value (if there is one).

doc is the document identifier. By default the current document is assumed.

confirmDtdOverride prompts the user whether to override the DTD if the graphic entity is already declared in the DTD. A non-zero value will prompt the user.

inEntity signifies whether the declaration is within an entity. A non-zero value means the declaration is within an entity.

declare_notation

declare_notation (*name*[, *sysid*[, *pubid*[, *doc*[, *confirmDtdOverride*]]]])

This function allows you to declare a notation with a specific name.

name is the name of an existing notation. A leading `&` is optional.

sysid is the new system identifier value (if there is one).

pubid is the new public identifier value (if there is one).

doc is the document identifier. By default the current document is assumed.

confirmDtdOverride prompts the user whether to override the DTD if the notation is already declared in the DTD. A non-zero value will prompt the user.

declare_text_entity

declare_text_entity (*name*[, *text*[, *type*[, *doc*[, *confirmDtdOverride*[, *inEntity*]]]]])

This function allows you to declare a text entity with a specific name and a specific replacement value.

name is the name of an existing text entity. A leading `&` is optional.

text is what replaces the entity reference in the formatted document.

type is either `cdata` to declare a CDATA entity, or an empty string .

doc is the document identifier. By default the current document is assumed.

confirmDtdOverride prompts the user whether to override the DTD if the graphic entity is already declared in the DTD. A non-zero value will prompt the user.

inEntity signifies whether the declaration is within an entity. A non-zero value means the declaration is within an entity.

defined

defined(*name*)

This function returns 1 if the argument *name* is defined. This function may be used to test the existence of a scalar or array variable, array element, function or package name. For example:

```
defined($x) or defined("x")
```

returns 1 if `x` is a scalar or array variable,
`defined("a[]")`

returns 1 if `a` exists as an array variable,
`defined($a[e1])`

returns 1 if the expression `(e1 in $a)` is true,
`defined("f()")`

returns 1 if `f` is a defined function,
`defined("p::")`

returns 1 if `p` is a loaded package.

Variable and function names may be prefixed with a package name, for example,
```
defined(main::x)
defined("utils::f()")
```

delete

`delete``(`*`name`*`)`

In this function, the argument *name* is a scalar or an array variable name or array element. If *name* is a scalar variable or array element, it is deleted and its value is returned. If *name* is an array variable, the contents of the array are deleted and the array name is removed from the symbol table. A null string is returned in this case. If the array name is a function formal parameter, then only the contents are deleted. This is useful for functions which take an array name as a parameter and fill it as a side effect. The argument to `delete` must not be a local variable or scalar function argument.

Example
```
val = delete(stack[p++])
```

delete_filep_entity

`delete_filep_entity` `(`*`name`*`[, `*`doc`*`])`

This function pernits you to both remove a reference to a file parameter entity (external parameter entity) and undeclare it from the document's internal subset.

name is the name for the entity. A leading percent sign (%) is optional.

doc is the document identifier. If *doc* is not supplied, the current document is assumed (default).

To ensure the file parameter's declarations are removed from the document's declaration. save the document then choose **File ► Revert to Saved**.

delete_markup_valid

delete_markup_valid ([*oid*[, *pos*[, *content*]]])

This function returns 1 if the markup identified by *oid* and *pos* can be deleted without context errors. It returns 0 if context errors would result if the markup was deleted.

All markup except included marked sections can be identified solely by *oid*. Both *oid* and *pos* must be given to specify an included marked section.

If *oid* is omitted, then `oid_caret` is used.

If *content* is given and non-zero, then this function returns `1` when no context errors would occur if both the markup and its content are deleted.

When testing whether markup can be deleted without context errors, this function will not do context transformations (such as tag substitutions, insert around to fix, insert required tags, remove redundant tags, and so on).

detail_tag

detail_tag (*tagname*[, *expand*[, *doc*]])

This function changes the detailing for all occurrences of the element named *tagname* in the document specified by *doc*. If *doc* is omitted or `0`, the current document is used.

The *expand* parameter, if given and non-zero, causes the element and its contents to be displayed. If *expand* is omitted or `0`, all occurrences of *tagname* are collapsed (detailed).

If *tagname* is a null string, `detail_tag` expands all elements in the document. In this case, the *expand* parameter is ignored.

dimen_convert

dimen_convert (*dimen*)

This function returns a string with *dimen* converted to inches. *dimen* must be a double-quoted string containing a number followed by a unit abbreviation. Valid abbreviations are: `in` (inches), `cm` (centimeters), `mm` (millimeters), `pt` (points), `pi` (picas), `px` (pixels), and `pc` (picas). `dimen_convert` returns a null string if *dimen* is not one of the valid abbreviations.

Examples:

- `dimen_convert("2cm")` returns `0.79in`
- `dimen_convert("25.5pc")` returns `4.23in`

direction

direction ([*oid*[, *pos*]])

This function returns 1 if the location indicated by *oid* and *pos* has a right-to-left directionality. It returns −1 if the location indicated by *oid* and *pos* has a left-to-right directionality.

If *oid* is omitted, the function uses oid_caret on page 430.

If *pos* is omitted, the function assumes 0.

If *oid* is invalid, the function returns 0.

dirname

dirname(*path*)

This function returns all but the last level of the path name in *path*. If *path* does not contain any path name components, the function returns the null string. dirname includes a trailing path separator as the last character of the string to permit easy construction of path names, for example,

Example

```
dirname(doc_path(doc)) . "test.sgm"
```

disable_windows

ret=**disable_windows** (*disable*[, *xid*])

Disables or re-enables user input from all windows (optionally skipping one window).

- *disable* — Boolean. When true, user input from all windows is disabled. When false, user input is enabled.
- *xid* — The native window system window handle (as would be returned by the function window_xid) of a window to be skipped. If *xid* is omitted or invalid, all windows are affected by disable_windows.

Caution

Use this function with extreme care. An application should only disable all windows if it has an alternative method of communicating with Arbortext Editor (for example, as when Arbortext Editor is acting as a server).

The call disable_windows(1) is equivalent to set userinput=off, and disable_windows(0) is equivalent to set userinput=on.

dita_doc_show_rm_tab

dita_doc_show_rm_tab (*tabID*[, *doc*])

This function displays the **Resource Manager** tab specified by the *tabID* parameter for the DITA document specified by the *doc* parameter. If no document is specified, the current document is used.

The following values are valid for the *tabID* parameter:

tabID 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
`new_topic_tab`	New Topic	DITA maps
`keydef_tab`	Key Definition	DITA maps
`xinclude_tab`	Inclusion	DITA topics and maps

dita_rde_xsd_from_map

dita_rde_xsd_from_map(*doc*[, *file*])

This function generates a resolved document for editing (`rdedit`) XML schema that includes all of the DITA topic document types referenced from the given DITA map. If the `ditaincludemapsinrde` preference is set to `on`, DITA map document types are also included.

The *doc* parameter is the DITA map to use as the basis of the new `rdedit` schema. The *file* parameter is the path to the location where you want to write the new schema. The function returns the full path to the generated file.

If *file* is a null string or omitted, a temporary `rdedit` schema is generated and will be deleted at the end of the current Arbortext Editor session.

dita_rds_dtd_from_map

dita_rds_dtd_from_map(*doc*[, *file*])

This function generates a resolved document for styling (`rdstyle`) DTD that includes all of the DITA maps and topic document types referenced from the given DITA map. The *doc* parameter is the DITA map to use as the basis of the

new `rdstyle` DTD. The *file* parameter is the path to the location where you want to write the new DTD. The function returns the full path to the generated file.

If *file* is a null string or omitted, a temporary `rdstyle` DTD is generated and will be deleted at the end of the current Arbortext Editor session.

dita_reset_rm_state

`dita_reset_rm_state` `()`

This function clears all persistent user preferences used to maintain or customize the **Resource Manager** state. The function also clears the current session's **Resource Manager** navigation history.

dita_rm_export_favorites

`dita_rm_export_favorites``(`*`file`*`)`

This function exports the current **Resource Manager** favorites list to the specified *file*. The function returns `1` on success or `0` on failure.

dita_rm_import_favorites

`dita_rm_import_favorites``(`*`file`*`[, ,`*`merge`*`])`

This function imports a **Resource Manager** favorites list from the specified *file*. If the optional *merge* parameter is set to `0`, the current favorites list is replaced by the imported list. If *merge* is omitted or set to any non-zero value, the imported list is appended to the current list of favorites. In this case, any duplicate entries are not added to the list.

The function returns `1` on success or `0` on failure.

dita_show_rm_tab

`dita_show_rm_tab` `(`*`tabID`*`[, `*`win`*`])`

This function displays the **Resource Manager** tab specified by the *tabID* parameter for the window specified by the *window* parameter. If no window is specified, the current window is used.

The following values are valid for the *tabID* parameter:

tabID 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
`new_topic_tab`	New Topic	DITA maps
`keydef_tab`	Key Definition	DITA maps
`xinclude_tab`	Inclusion	DITA topics and maps

ditaref_relative_path

`newpath=`**`ditaref_relative_path`**`(`*`path`*`[, `*`doc`*`])`

This function converts the absolute path name given by *path* into a relative path name if possible. If the file referenced by *path* is in or below the same directory as the document *doc*, a path name relative to the document directory is returned. If the file is located within a directory given by the `ditapath` Advanced Preference or `set` option, a path name relative to that directory is returned. Otherwise, the original input path name is returned.

If *doc* is zero or omitted, the current document is used.

This function is used by Arbortext Editor when storing file references in DITA documents.

ditaref_resolve

`path=`**`ditaref_resolve`** `(`*`path`*`, `*`oid`*`[, `*`attrname`*`])`

This function resolves a reference to content in a DITA document and returns the path where the referenced content is located.

- The *path* parameter is the file path or file name you want to resolve. The value of *path* can be a file name, an absolute or relative path, a URI, or a Logical ID. If the value of *path* is a relative path, it is resolved against the following directories in this order:
  - The base URI for *oid*, if any.
  - The directories in the DITA references path, if any.
  - The URI entries in the `catalog` path.

The URI entries in the `catalog` path are also checked for the *path* value in the following cases:

- The *path* value is an absolute file path to a file that does not exist.
- The *path* value is a content management system Logical ID or fully qualified URL.

If you want to have *path* resolved against the document directory and the DITA references path (shown in the **File Locations** category of the **Preferences** dialog box), but not the base URI, use the *oid* of the base element in the document. If you want to have *path* resolved against the values in the DITA references path, but not the base URI or the document directory, use `oid_null()` as the value of *oid*. The symbolic parameter `%^` in the DITA references path will also prevent the base URI and document directory from being used to resolve *path*.

- The *oid* parameter specifies the context in which *path* should be resolved. The context includes any base URI associated with the *oid* and the document directory for the document that contains the *oid*.
- The optional *attrname* parameter is the name of the attribute that contains the *path* value being resolved.

The returned path is the resolved path name and is usually an absolute path. If *path* is a relative path and cannot be resolved, the returned path is one of the following values:

- The absolute path for *path* in the base URI directory, if one exists.
- The absolute path for *path* in the document directory, if one exists.
- The value of *path*, if there is no base URI or document directory.
- An empty string, if `oid_null()` is used as the value of *oid*.

division_heading_tag

`division_heading_tag` (*`tagname`*`[,` *`doc`*`]`)

This function returns `1` (true) if the tag specified by *tagname* is declared as a division heading in the `.dcf` file for the document type associated with *doc*. It returns 0 (false) if it isn't. If *doc* is omitted or `0`, the current document is used.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

division_tag

division_tag (*tagname*[, *doc*[, *primary*]])

This function returns 1 (true) if the tag specified by *tagname* is declared as a division in the document type configuration (.dcf) file for the document type associated with *doc*. If *doc* is omitted or 0, the current document is used. If *primary* is specified and equal to 1, the function returns 1 (true) only if the *tagname* is also declared as a primary division in the .dcf file.

> **Note**
>
> If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

dl_builtin_addr

dl_builtin_addr (*fname*)

This function returns the address of specified built-in Arbortext Editor function. This address may be passed to a dynamically loaded function with dl_call to allow such functions to call back into Arbortext Editor to request various services, in particular, to execute and evaluate ACL commands and expressions.

The parameter *fname* may be one of the following:

acl_exec

Returns the address of the ACL interpreter. This is an interface to the ACL function exec. The built-in function has the signature:

```
typedef unsigned int (*ACLEXEC) (char *cmd);
```

where cmd is the ACL command to execute. The return value is the command status variable *main::status*. The command string is interpreted as a multi-byte string.

acl_execU

Same as acl_exec except the input string is a Unicode string.

acl_eval

Returns the address of the ACL expression evaluator. This is an interface to the ACL function eval. The built-in function has the signature:

```
typedef char* (*ACLEVAL) (char* expr, char* buf, int bufsize);
```

where expr is the string to evaluate, buf is an output buffer to receive the result of the expression as a string, and bufsize is the maximum size of the buffer. At most bufsize bytes will be written to buf.

The result will not be null-terminated if it is longer than `bufsize`-1 bytes. All strings are interpreted as multi-byte strings.

You can retrieve the value of an ACL variable by passing its name as the `expr` argument.

acl_evalU

Same as `acl_eval` except the strings are Unicode, (that is, the `char*` types in the `acl_eval` example are replaced by unsigned `short*`.

application

Returns the address of the C++ binding for the `Application` interface.

The function returns 0 if *fname* is not recognized.

> **Note**
>
> The function call `dl_builtin_addr('acl_exec')` replaces `interpreter_addr`.

The following is an example of some C functions that might be part of a dynamically loaded library.

```
 typedef unsigned int (*CMDFUNC) (char * acl_string);
 CMDFUNC acl_execp;
 void set_acl_addr(void *addr)
 {
 acl_execp = (CMDFUNC) addr;
}
 int aclcmd(char *str)
 {
 if (acl_execp)
 return (*acl_execp)(str);
 fprintf(stderr, "set_acl_addr() must be called first.\n");
 return 0;
}
```

The interpreter address is passed to the library as follows:

```
local set_acl_addr = dl_find(hdll, "set_acl_addr");
dl_call(set_acl_addr, dl_builtin_addr("acl_exec"));
```

dl_call

dl_call(*ref*[, *arg1*[, *argn*]])

This function calls a function in a dynamically loaded library. The *ref* parameter is a reference to a symbol returned by a previous call to `dl_find`. The remaining arguments, if any, are the parameters passed to the dynamically loaded function.

Numeric arguments are passed by value.

Strings are passed by reference. If the dynamic library being called expects 8-bit string parameters, `dl_call` will convert the string to 8-bit before it passes it to the library. If the library expects 16-bit Unicode strings, the string parameter is passed unchanged. Return values are not converted in this way. If necessary, the ACL code that called the library can use `unpack` to extract a string from a return value.

It is possible to pass a pointer to a C-style structure by using the `pack` function. See the example that follows.

The return value from `dl_call` is the value returned by the called function. If the return value is a scalar value, it may be used directly. If the return value is a pointer, then `unpack` must be used to convert the pointer into a string or extract parts of a structure.

The example that follows shows how to use `unpack` to unreference a pointer returned by a DLL function called with `dl_call`. In this case, the Windows API `GetCommandLine`, defined in `kernel32.dll` as "GetCommandLineA" — the ANSI version — returns a pointer to a null terminated string. `unpack` creates a copy of the string to be assigned to the variable `cmdline` so changing `cmdline` later does not affect the memory returned by `GetCommandLine`. Note that the return value is tested for null explicitly in this example, even though `unpack` can tolerate a null pointer and generate a null string.

```
dll = dl_load("kernel32")
hGetCommandLine = dl_find(dll, "GetCommandLineA")
ptr = dl_call(hGetCommandLine)
cmdline = ptr ? unpack("p", ptr) : ""
```

An example of passing a structure to a dynamically loaded function follows. In this case, the Windows API `GetSystemTime` is called, which takes a pointer to a `SYSTEMTIME` structure that has 8 fields of type `WORD` (short). The `pack` function creates this structure and `unpack` is used to convert the result into an array and to extract the member corresponding to the day of the week.

```
function GetDayOfWeek()
{
 local dll = dl_load("kernel32")
 if (!dll) {
 response("Couldn't load kernel32")
 return -1
}
 local GetSystemTime = dl_find(dll, "GetSystemTime")
 if (!GetSystemTime) {
 response("Couldn't find GetSystemTime")
 dl_unload(dll)
 return -1
}
 local timebuf = pack("s8")
 dl_call(GetSystemTime, timebuf)
 dl_unload(dll)
 local systime[]
```

```
 unpack("s8", timebuf, systime)
 return systime[3]; # wDayOfWeek
}
```

Since only a single field of the structure is used, the `unpack` call in the above example could be written more simply as

```
return unpack("x4s", timebuf)
```

which would replace the last three lines of the function.

The following example calls the Windows API `FindExecutable` to find the executable associated with a specified file name:

```
function find_exe(file, dir="")
{
 local dll = dl_load("shell32")
 if (!dll) {
 response("Couldn't load shell32")
 return "";
}
 local _FindExecutable = dl_find(dll, "FindExecutableA")
 if (!_FindExecutable) {
 response("Couldn't find FindExecutable")
 dl_unload(dll)
 return ""
}
 local buf = pack("a255")
 local h = dl_call(_FindExecutable, file, dir, buf)
 dl_unload(dll)
 if (h <= 32) {
 response("FindExecutable failed, return " . n)
 return ""
}
 return unpack("A*", buf)
}
```

A sample call follows:

```
function shell_exe(filename, dir="")
{
 local exename = find_exe(filename, dir)
 if (exename != "") {
 sh $exename $filename &; # launch program
}
}
```

Returning Input Pointers as Results

An entry point called by `dl_call` should not return one of its parameters as its result if the parameter is, or could be, a pointer to a string. Consider the following ACL code:

```
result = dl_call(do_nothing, "a string")
```

```
result = unpack("*a*", result)
```
Assume that the `do_nothing` DLL entry point is the following:
```
char *do_nothing(char *pChar)
{
 return pChar;
}
```
This code will fail if the library containing `do_nothing` expects 8–bit strings. In this case `dl_call` will make an 8–bit copy of the parameter *a string*, and release that memory as soon as the DLL entry point returns. When the call to `unpack` executes, the pointer returned by the DLL points to the released memory.

One possible way to write the entry point would be the following:

```
char *do_nothing(char *pChar)
{
 return strdup(pChar);
}
```
The only caveat to this method is that the DLL will need to release the string memory at some point, perhaps when it is passed to another entry point.

dl_error

dl_error ()

This function returns an error message describing the reason the last call to `dl_load`, `dl_find`, or `dl_unload` failed. It returns null if the last call succeeded or if the system does not support an error message or code for failed dynamic load calls.

dl_find

dl_find(*h*, *symbol*)

This function returns the address of the exported dynamic (shared) library function specified by *symbol*. The parameter *h* is the identifier of the dynamic library as returned by a previous call to `dl_load`. The address returned is an identifier that is passed to `dl_call` to actually invoke the function. `dl_find` returns `0` if the symbol cannot be found. The function `dl_error`, if supported by the operating system, may be called to obtain a message describing the reason for failure.

An example of an Arbortext Command Language interface to the Windows API `GetProfileString` follows.

```
function GetProfileString(section, key)
{
 local dll = dl_load("kernel32")
 if (!dll) {
```

```
 response("Couldn't load kernel32")
 return "";
}
 local _GetProfileString = dl_find(dll, \
 "GetProfileStringA")
 if (!_GetProfileString) {
 response("Couldn't find GetProfileStringA")
 dl_unload(dll)
 return ""
}
 local buf = pack("a255"); # get a return buffer
 dl_call(_GetProfileString, section, key, "", buf, 255)
 dl_unload(dll)
 return unpack("A*", buf); # trim trailing nulls
}
```

A sample call would be:

```
country = GetProfileString("Intl", "sCountry")
```

If this function is frequently called, or if other APIs from the same DLL are used, it would be better to load the DLL only once and remember the library identifier and function reference in global variables.

See `dl_call` for additional examples.

dl_load

dl_load(*file*[, *world*[, *type*]])

This function loads the dynamic (or shared) library specified by the path name *file*. `dl_load` returns an identifier which can be used in subsequent calls to `dl_find` and `dl_unload`. It returns 0 if the library could not be loaded. The function `dl_error` can be used on some systems to retrieve a message describing the nature of the error.

The library identifier returned by `dl_load` can be passed to the function `dl_find` to get the address of a function in the dynamic library to be called using `dl_call`.

The optional parameter *world* is ignored except on Solaris systems where it allows the shared library to reference symbols that are part of Arbortext Editor (if set to one (`1`). If omitted, it defaults to zero (`0`).

The optional parameter *type* specifies whether the dynamic library expects 16-bit Unicode strings or 8-bit strings in parameters and return values. A *type* of one (`1`) specifies Unicode strings; zero (`0`) specifies 8-bit strings. If omitted, it defaults to zero (`0`).

`dl_load` is an interface to the Windows API `LoadLibrary` and follows the same rules if file does not specify a path name. Because Arbortext Editor is a Win32 executable, the DLL loaded by `dl_load` must also be a 32-bit library. As with standard Windows APIs, functions exported by the DLL must be declared with the "standard" calling convention, that is, using the keyword `__stdcall`.

See `dl_find` and `dl_call` for examples.

dl_unload

`dl_unload` (*`h`*)

This function unloads the dynamic library specified by the identifier *h*, which is a return value from a previous call to `dl_load`. Any function references returned by `dl_find` for this library become invalid after `dl_unload` is executed. Note that the operating system normally uses a reference count to manage dynamically loaded libraries, so `dl_unload` may not actually unload the library. However, the handle *h* will no longer be valid after the call to `dl_unload`.

Example

```
dl_unload(hDll)
```

dlgitem_activate

`dlgitem_activate` (*`window`*, *`dlgitem`*)

Ensures that *dlgitem* is active when window is open. An active dialog item responds to user activity such as key strokes and mouse clicks.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the ACTIVE attribute of the dialog item with a non-zero value.

Any necessary display updates will be scheduled but not necessarily completed when this functions returns.

Example

```
$ret = dlgitem_activate($win, "Salary")
```

Activates the `Salary` dialog item.

dlgitem_collapse

`dlgitem_collapse` (*`window`*, *`dlgitem`*, *`listtag`*, *`row`*)

Collapses the list at the given point in the given list tag, within the dialog item *dlgitem* in *window*. If successful, this function returns a one (1). If the function fails, it returns a zero (0).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *listtag* parameter instructs the function to get the sublist for this list tag. The *row* parameter specifies which row of the sublist to collapse.

Notes

This function affects what is seen on the display; it has no effect on any data that is visible to the application.

Example

```
$top = dlgitem_get_listtag($win, "Hierarchy")
$ret = dlgitem_collapse($win, "Hierarchy", $top, 0)
```

Collapses the child of the first entry in the `Hierarchy` list.

dlgitem_deactivate

`dlgitem_deactivate` (*`window`*, *`dlgitem`*)

Ensures that *dlgitem* is not active when window is open. An inactive dialog item does not respond to user activity such as key strokes and mouse clicks. If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the ACTIVE attribute of the dialog item with a zero (0) value.

Any necessary display updates will be scheduled but not necessarily completed when this functions returns.

Example

```
$ret = dlgitem_deactivate($win, "Salary")
```

Deactivates the `Salary` dialog item.

dlgitem_display

`dlgitem_display` (*`window`*, *`dlgitem`*)

Ensures that the *dlgitem* is visible when *window* is open. This function is specifically useful on the toolbar, since it automatically invokes a re-display of the toolbar, shifting the other toolbar buttons to the right (if necessary) to create space for the button displayed (if the window is already open, you will see a momentary flash as this re-display takes place).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example

```
$ret = dlgitem_display($win, "Toolbar_Equation")
```

Shows the **Equation** toolbar button.

dlgitem_dropdown_list

dlgitem_dropdown_list(*window*, *dlgitem*)

This function accesses the dropdown lists in the Arbortext Editor toolbar. The *window* parameter is a window identifier and indicates in which window you want the dropdown list to be activated. *dlgitem* is the value of the control's *id* attribute. For example, the following command would activate the Insert Markup dropdown list.

```
$ret = dlgitem_dropdown_list(current_window(), \
"Toolbar_InsertMarkupDrop")
```

The function returns a one (`1`) on success and returns a zero (`0`) if the *dlgitem* was not found or is not a dropdown list.

The following is a list of the available dropdown lists and the values for *dlgitem* that will activate them.

Dropdown List	**Value for *dlgitem***
Insert Markup	`Toolbar_InsertMarkupDrop`
Insert Symbol	`Toolbar_InsertCharEntityDrop`
Insert Text Entity	`Toolbar_InsertTextEntityDrop`
Insert File Entity	`Toolbar_InsertFileEntityDrop`
Bookmarks	`Toolbar_BookmarkDrop`
Collapse/Expand	`Toolbar_CollapseExpandDrop`
Cell Shading	`Toolbar_CellShadeDrop`

dlgitem_ensure_listtag_visible

dlgitem_ensure_listtag_visible(*window*, *dlgitem*, *listtag*)

Instructs Arbortext Editor to scroll the tree control specified by *dlgitem* so that the treenode *listtag* is visible.

- *window* — The window identifier of the window containing the tree.
- *dlgitem* — The value of the tree control's *id* attribute.
- *listtag* — The list tag for which the function returns the sublist.

If *listtag* refers to a valid list tag, `1` is returned. Otherwise, the function returns `0`.

dlgitem_ensure_table_visible_at

dlgitem_ensure_table_visible_at(*window*, *dlgitem*, *row*)

Instructs Arbortext Editor to scroll the window so that the row *row* is displayed.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *row* — The 1–based index of the row to be displayed.

If *dlgitem* does not refer to a table, or if *row* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_exists

dlgitem_exists (*window*, *dlgitem*)

If the specified control exists in the window, the function returns `1`. Otherwise, `dlgitem_exists` returns `0`.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

dlgitem_expand

dlgitem_expand (*window*, *dlgitem*, *listtag*, *row*)

Expands the list at the given point in the given list tag (in outline lists), within the dialog item *dlgitem* in *window*. If successful, this function returns a one (1). If the function fails, it returns a zero (0).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *listtag* parameter instructs the function to get the sublist for this list tag. The *row* parameter specifies which row of the sublist to expand.

The EXPAND callback will be called to request more rows.

Example

```
$top = dlgitem_get_listtag($win, "Hierarchy")
$ret = dlgitem_expand($win, "Hierarchy", $top, 0)
```

Expands the child of the first entry in the `Hierarchy` list.

dlgitem_find_table_cell_in_column

`dlgitem_find_table_cell_in_column``(`*`window`*`, `*`dlgitem`*`, `*`column`*`, `*`text`*`)`

Searches a column in a table for the cell containing *text*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The 1–based index of the column to search.
- *text* — The content to search for.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `-1` is returned. If no cell in the column contains *text*, `0` is returned. Otherwise, the 1–based index of the first row containing *text* is returned.

dlgitem_get

`dlgitem_get` `(`*`window`*`, `*`dlgitem`*`, `*`attribute`*`)`

Returns the value of *attribute* for *dlgitem* within *window* as a string.

- *window* — A window identifier.
- *dlgitem* — The value of the control's *id* attribute.
- *attribute* — The attribute to get. For a list of attributes, refer to `dlgitem_set` on page 296.

If *window* is invalid, or if *dlgitem* does not exist within window, then $ERROR is set and NULL is returned.

dlgitem_get_active_at

`dlgitem_get_active_at``(`*`window`*`, `*`listtag`*`, `*`row`*`)`

Returns active state of the specific outline list entry *row* in the outline list *listtag* in *window*. If *listtag* does not refer to an outline list list tag, $ERROR is set and 0 is returned. Otherwise, the active state is returned (0 = not active, 1 = active).

- *window* — The window identifier.
- *listtag* — An outline list list tag (case sensitive).
- *row* — An offset into *listtag* (starting from one).

Example

```
$ret = dlgitem_get_active_at($win, "Hierarchy", 3)
```

Gets the active state of the third item in the outline list item `Hierarchy`.

dlgitem_get_all

dlgitem_get_all (*window*, *attribute*, *array*)

Fills *array*, an associative array, with the value of the *attribute* for all of the dialog items in *window*. If a particular dialog item does not have the attribute, no entry is made in the array. If no dialog items have the attribute, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. The *attribute* parameter specifies the attribute to get. The *values* parameter specifies the associative array to fill.

Example

```
$ret = dlgitem_get_all($win, "VALUE", $Values)
```

Returns the following array:

```
$Values["NameField"]: "Jeff"
$Values["AgeField"]: "35"
$Values["StateField"]: "MA"
```

dlgitem_get_appdata

dlgitem_get_appdata (*window*, *dlgitem*)

Returns the application-specific data associated with dialog item *dlgitem* in *window*. If successful, the function returns the application-specific data associated with the desired *dlgitem*. If there is no application-specific data for the desired *dlgitem*, the routine returns a NULL. If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), $ERROR is set and 0 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example

```
$ret = dlgitem_get_appdata($win, "Hierarchy")
```

Returns the string "TopOne" when used in conjunction with the sample code for `dlgitem_set_appdata`.

dlgitem_get_appdata_at

dlgitem_get_appdata_at(*window*, *listtag*, *row*)

Returns the user-defined data associated with the list entry *row* in the outline list *listtag* in *window*. If *dlgitem* contains no list or *row* is out of range of the list, the function returns an empty string. If the function fails, it returns a NULL.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list list tag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one).

Example

```
$ret = dlgitem_get_appdata_at($win, "Hierarchy", 3)
```

Gets the user-defined data associated with the third item in the outline list item `Hierarchy`.

dlgitem_get_background_at

dlgitem_get_background_at(*window*, *listtag*, *row*)

Returns the background color for the outline list tag identified by *listtag* in *window*. If *listtag* does not refer to an outline list tag, `$ERROR` is set and 0 is returned. If no color is specified at that location, an empty string is returned. Otherwise, a named color or an RGB specification preceded by # is returned.

window
the dialog window identifier

listtag
an outline list tag (case sensitive)

row
an offset into listtag(starting from `one`)

dlgitem_get_check_at

dlgitem_get_check_at (*window*, *listtag*, *row*)

Returns the check state of the specific checkable outline list entry *row* in the checkable outline list *listtag* in *window*. If *listtag* does not refer to a checkable outline list list tag, $ERROR is set and 0 is returned. Otherwise, the check state is returned.

Possible return values:

- 0 — list entry is not checked.
- 1 — list entry is checked.
- –1 — list entry contains intermediate check. This results when a list entry contains a sublisttag with some list entries checked, but not all.

The *window* parameter is a window identifier. The *listtag* parameter is a checkable outline list list tag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one).

Example

```
$ret = dlgitem_get_check_at($win, "Hierarchy", 3)
```

Returns the check state of the third item in the checkable outline list item `Hierarchy`.

dlgitem_get_focus

`ret` = **`dlgitem_get_focus`**(*`win`*)

Returns the tag name of the dialog item (case sensitive) that has keyboard focus for the window specified by *win*. If *win* is invalid, or if the keyboard focus has not been set for the window, then `$ERROR` is set and the empty string ("") is returned.

The *win* parameter is a window identifier.

Example

```
focused_item = dlgitem_get_focus($win)
```

dlgitem_get_foreground_at

`dlgitem_get_foreground_at`(*`window`*, *`listtag`*, *`row`*)

Returns the foreground color for the outline list tag identified by *listtag* in *window*. If *listtag* does not refer to an outline list tag, `$ERROR` is set and 0 is returned. If no color is specified at that location, an empty string is returned. Otherwise, a named color or an RGB specification preceded by # is returned.

window
: the dialog window identifier

listtag
: an outline list tag (case sensitive)

row
: an offset into listtag(starting from `one`)

dlgitem_get_list_array

dlgitem_get_list_array(*window*, *dlgitem*, *array*)

Returns the entire list of values in the list dialog item or list tag (in outline lists) identified by *dlgitem* in *window*. If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *array* parameter specifies the array for the returned values.

To gather only the selected items in a dialog list item, use the dlgitem_get_select_array on page 287 function.

Example

```
$ret = dlgitem_get_list_array($win, "Foliage", $Trees)
```

Gathers all the values in the *Foliage* dialog list item and places them in the array `$Trees`.

dlgitem_get_list_at

dlgitem_get_list_at (*window*, *dlgitem*, *index*)

Returns the value at index in the list of values in the list dialog item or list tag (in outline lists) identified by *dlgitem* in *window*. If *index* is invalid or if *dlgitem* does not refer to a dialog item or list tag (in outline lists), $ERROR is set and NULL is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *index* parameter specifies the index into list on *dlgitem*.

Example

```
$ret = dlgitem_get_list_at($win, "Languages", 1)
```

Returns the string "C" for the list used in the sample code for `dlgitem_set_list_at`.

dlgitem_get_list_count

dlgitem_get_list_count(*window*, *dlgitem*)

Returns the number of items in the list associated with *dlgitem* in *window*. Returns (-1) if the item does not have an associated list; *dlgitem* may also refer to a list tag (in outline lists).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

Calling this function has the same effect as calling the `dlgitem_get` function on the LIST_COUNT attribute of the dialog item.

When used on the root of an outline list item, returns the count of the top-level list tag of the list.

Example

```
$ret = dlgitem_get_list_count($win, "Children")
```

Returns the number of items in the `Children` dialog item.

dlgitem_get_listtag

`dlgitem_get_listtag` (*`window`*, *`dlgitem`*)

Returns the top-level list tag (in outline lists) for the dialog item, or sets $ERROR and returns NULL.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

If the dialog item is an outline list item but has no list tag, a list tag will be created and returned.

Example

```
$ret = dlgitem_get_listtag($win, "Hierarchy")
```

Obtains the list tag of the dialog item named Hierarchy.

dlgitem_get_listtag_by_appdata

`dlgitem_get_listtag_by_appdata`(*`window`*, *`dlgitem`*, *`appdata`*)

Searches the tree specified by *dlgitem* and returns the list tag (in outline lists) of the tree node whose application-specific *appdata* is the same as the *appdata* specified in the function call.

- *window* — The window identifier of the window containing the tree.
- *dlgitem* — The value of the control's *id* attribute.
- *appdata* — The *appdata* attribute of a tree node.

dlgitem_get_mnemonic

`dlgitem_get_mnemonic` (*`win`*, *`dlgitem`*)

Returns the letter that is the mnemonic for the dialog item specified by *dlgitem* in the window specified by *win*. If *win* or *dlgitem* are invalid, then `$ERROR` is set and the empty string ("") is returned. If the specified *dlgitem* has no mnemonic, the empty string ("") is returned.

The *win* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example

```
mnem_letter = dlgitem_get_mnemonic($win, "matchMarkup")
```

dlgitem_get_select_array

dlgitem_get_select_array(*window*, *dlgitem*, *array*)

Returns an array of selected values in the list dialog item or list tag (in outline lists) identified by *dlgitem* in *window*. If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), *$ERROR* is set and 0 is returned. Otherwise, 1 is returned.

- *window* — A window identifier.
- *dlgitem* — The value of the control's *id* attribute.
- *array* — Specifies the array for the returned values. If a XUI 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.

To gather all the items in a dialog list item, regardless of selection, use the `dlgitem_get_list_array` on page 285 function.

Example

Consider a *Trees* list dialog list item containing the following tree names:

```
Oak
Maple
Gum
Gingko
Birch
Aspen
```

If a user selected the `Oak` and `Gum` list items, the following expression would gather the selected items and place them in an array called *$SelectTrees*.

```
$ret = dlgitem_get_select_array($win, "Trees", $SelectTrees)
```

The *$SelectTrees* array would contain the following values:

```
Values[1]: "Oak"
Values[2]: "Gum"
```

dlgitem_get_selected_appdata

dlgitem_get_selected_appdata(*window*, *dlgitem*)

Returns the user-defined data associated with item *dlgitem* in *window*. If *dlgitem* contains no selection or the selected item contains no data, the function returns an empty string. If the function fails, it returns a NULL.

With XUI tree controls, 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.

- *window* — A window identifier.
- *dlgitem* — The value of the control's *id* attribute.

Example

```
$top = dlgitem_get_selected_appdata($win, "Hierarchy")
```

Returns the selected item in the `Hierarchy` list.

dlgitem_get_selected_listtag

dlgitem_get_selected_listtag(*window*, *dlgitem*)

Returns the list tag of the selected tree node in the tree control specified by *dlgitem* 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.

- *window* — The window identifier of the window containing the tree.
- *dlgitem* — The value of the control's *id* attribute.

dlgitem_get_selected_listtag_array

dlgitem_get_selected_listtag_array(*window*, *dlgitem*, *array*)

Returns an array containing the list tags of all selected nodes in the tree control specified by *dlgitem*. 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.

- *window* — The window identifier of the window containing the tree.
- *dlgitem* — The value of the control's *id* attribute.
- *array* — The array of returned values.

dlgitem_get_sublisttag

dlgitem_get_sublisttag(*window*, *dlgitem*, *listtag*, *row*[, *nobranchconv*])

Returns the list tag of the sublist of the given list tag (in outline lists), or optionally inserts and returns a new, empty sublist as a child of the given row of the existing list tag within item *dlgitem* in *window*. If the function fails, it returns an empty string.

- *window* — The window identifier.
- *dlgitem* — The value of the control's *id* attribute.
- *listtag* — The list tag for which the function returns the sublist.
- *row* — Specifies which row of the sublist to return.
- *nobranchconv* — Optional. If `1`, and the tree node specified is a leaf, the leaf will not be converted to a branch, allowing users to get the list tag of a leaf without changing anything. If *nobranchconv* is `0`, the leaf is converted to a branch. The default is `0`

Example

```
$top = dlgitem_get_listtag($win, "Hierarchy")
$ret = dlgitem_get_sublisttag($win, "Hierarchy", $Top,0)
```

Returns the child of the first entry in the `Hierarchy` list.

dlgitem_get_table_cell_at

dlgitem_get_table_cell_at(*window*, *dlgitem*, *row*, *column*)

Returns the value in the cell specified by the intersection of *row* and *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *row* — The `1`-based index of the row containing the cell.
- *column* — The `1`-based index of the column containing the cell.

If *dlgitem* does not refer to a table, or if *column* or *row* are illegal values, *$ERROR* is set and `NULL` is returned. Otherwise, the value of the cell is returned.

dlgitem_get_table_column_align

dlgitem_get_table_column_align(*window*, *dlgitem*, *column*)

Returns the alignment information of the column specified by *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The `1`-based index of the column. If the table has fewer columns that the value of *column*, the table is extended.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `NULL` is returned. Otherwise, the alignment value (`LEFT`, `RIGHT`, or `CENTER`) is returned.

dlgitem_get_table_column_count

`dlgitem_get_table_column_count``(`*`window`*`, `*`dlgitem`*`)`

Returns the number of columns in a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the table's *id* attribute.

If *dlgitem* does not refer to a table, *$ERROR* is set and `-1` is returned. Otherwise, the number of columns in the table is returned.

dlgitem_get_table_column_header_at

`dlgitem_get_table_column_header_at``(`*`window`*`, `*`dlgitem`*`, `*`column`*`)`

Returns the header label of the column specified by *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The `1`-based index of the column. If the table fewer columns than the value of *column*, the table is extended.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `NULL` is returned. Otherwise, the header label is returned.

dlgitem_get_table_row_count

`dlgitem_get_table_row_count``(`*`window`*`, `*`dlgitem`*`)`

Returns the number of rows in a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the table's *id* attribute.

If *dlgitem* does not refer to a table, *$ERROR* is set and `-1` is returned. Otherwise, the number of rows in the table is returned.

dlgitem_get_table_selection

dlgitem_get_table_selection(*window*, *dlgitem*[, *column*])

Returns the selected row when the selection type of the table control is row selection. If the selection type is cell selection, `dlgitem_get_table_selection` returns the row of the selected cell. The function returns `-1` if the table does not exist. The function returns `0` if nothing is selected.

- *window* — The window identifier of the window containing the table control.
- *dlgitem* — The value of the table control *id* attribute.
- *column* — If the table control's selection type is cell selection, this output argument stores the column of the selected cell.

dlgitem_get_table_sort

dlgitem_get_table_sort(*window*, *dlgitem*[, *sortorder*])

Returns the index of the column on which the table is sorted.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *sortorder* — The value of the *sortorder* is set by the function. It reflects the current sort order of the current sorted column. If the value of the *sortorder* is `0`, the column is sorted in ascending order. Otherwise, the column is sorted in descending order.

If *dlgitem* does not refer to a table, *$ERROR* is set and `-1` is returned. If the table is not sorted, `0` is returned. Otherwise, the index of the column on which the table is sorted is returned.

dlgitem_get_value

dlgitem_get_value (*window*, *dlgitem*)

Returns the VALUE attribute of *dlgitem* within *window* as a string. If *window* is invalid, or if *dlgitem* does not exist within window, then $ERROR is set and NULL is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Examples

```
$ret = dlgitem_get_value($win, "TextField")
```

Notebooks can be queried. By using this function on a `gpNotebook` item, one can query which tab is active.

```
LINK_NOTEBOOK = "gpNotebookTagName"
# tab_value is an integer, which represents the position
# of the tabs by counting from left to right within
# the notebook.
function query_activate_tab(win) {
 return dlgitem_get_value(win, LINK_NOTEBOOK)
}
```

dlgitem_hide

dlgitem_hide (*window*, *dlgitem*)

Ensures that *dlgitem* is not visible when *window* is open.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the VISIBLE attribute of the dialog item with a zero (0) value.

Any necessary display updates will be scheduled but not necessarily completed when this functions returns.

Example

```
$ret = dlgitem_hide($win, "Salary")
```

Hides the `Salary` dialog item.

dlgitem_insert_list_at

dlgitem_insert_list_at(*window*, *listtag*, *row*, *value*)

For the location specified by *listtag* and *row*, performs one of the following actions:

- Inserts a new tree node in a tree.
- Inserts a new list item in a listbox or combobox.
- Inserts a new row in a tablecontrol.

The label of the new node, list item, or row is specified by *value*.

- *window* — The window identifier of the window containing the tree, listbox, combobox, or tablecontrol.
- *listtag* — The `list` tag (identifier of the node) of the tree branch or the value of the control's *id* attribute.

- *row* — The index of the tree node in the tree branch, list item in the list box or combobox, or row in the tablecontrol.
- *value* — The string to insert.

dlgitem_insert_table_column_at

dlgitem_insert_table_column_at(*window*, *dlgitem*, *column*[, *header*])

Inserts a new column in a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The 1-based index of the new column. If the table has fewer columns than the value of *column*, the table is extended.
- *header* — Optional. The header label of the new column.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_insert_table_row_at

dlgitem_insert_table_row_at(*window*, *dlgitem*, *row*[, *text*])

Inserts a new row in a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *row* — The 1-based index of the new row. If the table has fewer rows than the value of *row*, the table is extended.
- *text* — Optional. The content of the first column in the row.

If *dlgitem* does not refer to a table, or if *row* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_is_active

dlgitem_is_active (*window*, *dlgitem*)

This function tests whether the *dlgitem* in *window* is active/enabled or not. If the *dlgitem* is active/enabled, the function returns a one (1). If the *dlgitem* is inactive/disabled, the function returns a zero (0). If *window* is invalid, or if *dlgitem* does not exist within *window*, then the function returns -1.

Example

```
$ret = dlgitem_is_active($win, "TextField")
```

dlgitem_is_expandable

dlgitem_is_expandable(*window*, *dlgitem*, *listtag*, *row*)

Returns 1 if the given list entry has a sublist, or 0 if it does not.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *listtag* parameter instructs the function to get the sublist for this list tag. The *row* parameter specifies which row of the sublist to test.

Example

```
$top = dlgitem_get_listtag($win, "Hierarchy")
$ret = dlgitem_is_expandable($win, "Hierarchy", $top, 0)
```

Returns one (1) if the child of the first entry in the `Hierarchy` is expandable.

dlgitem_is_expanded

dlgitem_is_expanded (*window*, *listtag*, *row*)

Returns `true` if the tree node (specified by *listtag*) has children and the children are expanded. Otherwise, returns `false`.

- *window* — The window identifier of the window containing the tree.
- *listtag* — The `list` tag (identifier of the node) of the tree branch.
- *row* — The index of the tree node in the tree branch.

dlgitem_remove_list_at

dlgitem_remove_list_at(*window*, *listtag*, *row*)

For the location specified by *listtag* and *row*, performs one of the following actions:

- Removes a tree node in a tree.
- Removes a list item in a listbox or combobox.
- Removes a row in a tablecontrol.

The parameters have the following values:

- *window* — The window identifier of the window containing the tree, listbox, combobox, or tablecontrol.
- *listtag* — The `list` tag (identifier of the node) of the tree branch or the value of the control's *id* attribute.
- *row* — The index of the tree node in the tree branch, list item in the list box or combobox, or row in the tablecontrol.

dlgitem_remove_table_column_at

`dlgitem_remove_table_column_at``(`*`window`*`, `*`dlgitem`*`, `*`column`*`)`

Removes a column from a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The 1–based index of the column to be removed.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_remove_table_row_at

`dlgitem_remove_table_row_at``(`*`window`*`, `*`dlgitem`*`, `*`row`*`)`

Removes a row from a table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *row* — The 1-based index of the row to be removed.

If *dlgitem* does not refer to a table, or if *row* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_remove_toolbar

`dlgitem_remove_toolbar``(`*`window`*`, `*`dlgitem`*`)`

Turns off the display of the toolbar specified by *dlgitem*. If *dlgitem* does not refer to a toolbar, `$ERROR` is set and `0` is returned. Otherwise, `1` is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

dlgitem_select_list_at

dlgitem_select_list_at(*window*, *listtag*, *row*)

For the location specified by *listtag* and *row*, performs one of the following actions:

- Selects a list item in a list box or combo box.
- Selects a row in a table control.
- Selects a tree node in a tree. 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.

The parameters have the following values:

- *window* — The window identifier of the window containing the tree, listbox, combobox, or tablecontrol.
- *listtag* — The `list` tag (identifier of the node) of the tree branch or the value of the control's *id* attribute.
- *row* — The index of the tree node in the tree branch, list item in the list box or combo box, or row in the table control.

dlgitem_set

dlgitem_set (*window*, *dlgitem*, *attribute*[, *value*,...])

Sets the value of the attribute named *attribute* of *dlgitem* to *value*. If *window* is invalid, if *dlgitem* does not exist within *window*, if *attribute* is not a valid attribute given the type of *dlgitem*, or if *value* is not valid given the type of *attribute*, then $ERROR is set and `0` is returned. If successful, the function returns a one (`1`).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *attribute* parameter(s) specifies the actual attribute(s) to set. The *value* parameters contain string data values for attributes.

Any necessary display updates will be scheduled but not necessarily completed when this functions returns.

Example

```
$ret = dlgitem_set($win, "TextField", VALUE,"Machu Pichu")
```

Dialog Item Attributes

All dialog items have the following attributes:

Note

Attribute types are expressed as strings in ACL (for example, `ACTIVE`), and are not case sensitive.

- `ACTIVE` — An integer: non-zero if the dialog item is active, zero if the dialog item is not active. An active dialog item responds to user actions (mouse clicks and key strokes).
- `APPDATA` — A string: application-specific data that is passed to dialog item and outline list expansion callbacks.
- `BACKGROUND` — An integer: the red-green-blue value denoting the item's background color. The value is either a named color or an RGB specification preceded by `#`.
- `CALLBACKS` — A string: a comma-delimited list of callback for the item, in the order that the callbacks are used.
- `DLGITEM_PTR` — An integer: Returns the window pointer of the MFC dialog item.
- `EDITABLE` — An integer: non-zero if the dialog item accepts user modifications, zero if the dialog item does not accept user modifications.
- `FOREGROUND` — An integer: the red-green-blue value denoting the item's foreground color. The value is either a named color or an RGB specification preceded by `#`.
- `GEOMETRY` — A string. If *dlgitem* is the XUI ID of an existing control, the value of `geometry` is a string representing the screen geometry of the control. The string has the format *width* x *height* + *X* + *Y*, where *width* is the width of the control, *height* is the height of the control, *X* is the horizontal screen coordinate, and *Y* is the vertical screen coordinate. For example, `520x320+100+200`. If *dlgitem* is not the XUI ID of an existing control, `dlgitem_get` returns an empty string.

  geometry is read only and cannot be set using a call to `dlgitem_set`.
- `IMAGE` — A string: the name of an image to display within the dialog item.
- `sourcecontrol` — A string. If *dlgitem* is the XUI ID of a menu bar, a context menu, or a dropdown menu, the value of `sourcecontrol` is the XUI ID of the control currently associated with the menu. For a menu bar, `dlgitem_get` returns the ID of the menu bar. For context and dropdown menus, `dlgitem_get` returns the ID of the control for which the menu is posted (such as a tree, table, or button control). If this menu was posted using

the `menu_popup` ACL function, no control is associated with the menu and `dlgitem_get` returns an empty string.

sourcecontrol is read only and cannot be set using a call to `dlgitem_set`.

- `TITLE` — A string: the title of the dialog item. Not all dialog items display their title.
- `VALUE` — A string: the value of the dialog item.
- `VISIBLE` — An integer: non-zero if the dialog item is visible, zero if the dialog item is not visible. A visible dialog item is shown when its dialog is open.
- `WITHDRAW` — Makes the dialog item invisible when the window is open. This is useful on the toolbar, because it automatically resets the display of the toolbar, shifting the other toolbar buttons to fill the space left by a withdrawn button (if the window is already open, you will see a momentary flash as this re-display takes place).

Dialog Item Specific Attributes

Certain dialog items may have type-specific attributes in addition to the default attributes for dialog items:

Note

Attribute types are expressed as strings in ACL (for example, `"EXPAND"`), and are not case sensitive.

- `COLLAPSE` — A callback function that is called to when an *listtag* is collapsed in an outlist item.
- `ENABLE` — A callback function that is called to enable or disable menu items before a menu is opened.
- `EXPAND` — A callback function that is called when a *listtag* is expanded in an outlist item.
- `EXTRAIMGCB` — A callback function that is called when the extra image of an outlist item is clicked with the left mouse button.
- `KEYBOARD` — A callback function that is called when a keyboard key is pressed for an outlist item.
- `LIST_COUNT` — An integer for items which contain lists. Contains the

number of items in the list. Setting the value extends or truncates the list as appropriate.

- `ON` — An integer (zero or non-zero) representing the check status of a checkbox menu item or checkbox toolbar button.

dlgitem_set_active_at

dlgitem_set_active_at(*window*, *listtag*, *row*, *activestatus*)

Activates or deactivates the specific outline list entry *row* in the outline list *listtag* in *window*. If *listtag* does not refer to an outline list list tag, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list list tag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *activestatus* parameter sets the enabled/disabled status of the list entry (1=enabled; 0=disabled).

Example

```
$ret = dlgitem_set_active_at($win, "Hierarchy", 3, 0)
```

Deactivates the third item in the outline list item `Hierarchy`.

dlgitem_set_appdata

dlgitem_set_appdata (*window*, *dlgitem*, *appdata*)

Associates application-specific *appdata* with the dialog item specified *dlgitem* in *window*. If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *appdata* parameter the string of application data to associate with the dialog item.

Notes

The application data is passed to dialog item and outline list expansion callbacks.

Example

```
$ret = dlgitem_set_appdata($win, "Hierarchy", "TopOne")
```

Sets the application data of dialog item `Hierarchy` to `TopOne`.

dlgitem_set_appdata_at

dlgitem_set_appdata_at(*window*, *listtag*, *row*, *appdata*)

Associates application-specific *appdata* with the outline list entry *row* in the outline list *listtag* in *window*. If *listtag* does not refer to an outline list listtag, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list listtag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *appdata* parameter is the string of application data to associate with the list item.

Notes

The application data is passed to dialog item and outline list expansion callbacks.

Example

```
$ret = dlgitem_set_appdata_at($win, "Hierarchy", 3, "TopOne")
```

Sets the application data of the third item in the outline list item `Hierarchy` to `TopOne`.

dlgitem_set_background_at

`dlgitem_set_background_at``(`*`window`*`, `*`listtag`*`, `*`row`*`, `*`background`*`)`

Sets the background color for the outline list tag identified by *listtag* in *window*. If *listtag* does not refer to an outline list tag, `$ERROR` is set and 0 is returned. Otherwise, 1 is returned.

window
the dialog window identifier

listtag
an outline list tag (case sensitive)

row
an offset into listtag(starting from `one`)

background
a named color or an RGB specification preceded with #

dlgitem_set_check_at

`dlgitem_set_check_at` `(`*`window`*`, `*`listtag`*`, `*`row`*`, `*`checkstate`*`)`

Sets the check state of the specific checkable outline list entry *row* in the checkable outline list *listtag* in *window*. If *listtag* does not refer to a checkable outline list list tag, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. The *listtag* parameter is a checkable outline list listtag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *checkstate* parameter indicates the state the check box for the list entry should take. (0 indicates no check, 1 indicates a check).

Example

```
$ret = dlgitem_set_check_at($win, "Hierarchy", 3, 0)
```

Sets the check state for the third item in the checkable outline list item in `Hierarchy` to unchecked.

dlgitem_set_default_branch_image

`dlgitem_set_default_branch_image``(`*`window`*`, `*`dlgitem`*`, `*`image`*`)`

Sets the image resource *image* as the image to be used for all list entries containing sublisttags that are in a collapsed state in an outline list dialog item *dlgitem* (unless overridden by a specific branch image setting for a particular listtag list entry) in window *window*.

The *image* appears at the front of all the list entries that contain sublisttags that are collapsed. *image* cannot be defined by a XUI `<imagelist>` element. It must be defined as a standalone `<image>` element.

Use `dlgitem_set_default_branch_image` to set the default image for list items that have children. If you were to create a file browser using an outline list, you may use a closed folder icon as your default branch image. If *dlgitem* does not refer to an outline list item, `$ERROR` is set and zero (0) is returned. Otherwise, one (`1`) is returned.

window is a window identifier. *dlgitem* is the value of the control's *id* attribute. *image* specifies the *id* attribute of an image element in the XUI file. The image is used to replace the image at the front of all list entries containing sublists for outline list item *dlgitem*.

For example,

```
ret = dlgitem_set_default_branch_image(win, "OutlistItem", "Folder")
```

sets the image for all list entries containing sublists and are collapsed in the outline list item `OutlistItem` to the `Folder` image.

dlgitem_set_default_leaf_image

`dlgitem_set_default_leaf_image``(`*`window`*`, `*`dlgitem`*`, `*`image`*`)`

Sets *image* as the default image to be used for all list entries that do not contain sublists in an outline list dialog item *dlgitem* (unless overridden by a specific leaf image setting for a particular listtag list entry) in window *window*. The *image* appears at the front of all the list entries that do not contain sublists. If you were to

create a file browser using an outline list, you may use a file icon as a default leaf image. If *dlgitem* does not refer to an outline list item, `$ERROR` is set and zero (0) is returned. Otherwise, one (`1`) is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example
```
ret = dlgitem_set_default_leaf_image(win, "OutlistItem", "File")
```

sets the image for all list entries that do not contain sublists in the outline list item `OutlistItem` to the `File` image.

dlgitem_set_default_openbranch_image

dlgitem_set_default_openbranch_image(*window*, *dlgitem*, *image*)

Sets *image* as the image to be used for all list entries containing sublisttags that are in an expanded state in an outline list dialog item *dlgitem* (unless overridden by a specific open branch image setting for a particular listtag list entry) in window *window*. The *image* appears at the front of all the list entries that contain sublisttags that are expanded. Use `dlgitem_set_default_openbranch_image` to set the default image for list items that have children and are expanded. If you were to create a file browser using an outline list, you may use a open folder icon as your default open branch image. If *dlgitem* does not refer to an outline list item, `$ERROR` is set and zero (0) is returned. Otherwise, one (`1`) is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example:
```
ret = dlgitem_set_default_openbranch_image(win, "OutlistItem", "Openfolder")
```

This sets the image for all list entries containing sublists and that are expanded in the outline list item `OutlistItem` to the `Openfolder` image.

dlgitem_set_focus

dlgitem_set_focus (*win*, *dlgitem*)

Changes the current keyboard focus of the window specified by *win* to the dialog item specified by *dlgitem*. If *win* is invalid, or if *dlgitem* does not exist within *win*, 0 is returned. If successful, the function returns 1.

The *win* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Note that this function also changes the default focus item for the window so that the next time the window is opened, the specified *dlgitem* will receive focus. If the window is already open, the current focus item will be unfocused and new item will be given focus.

Example

```
$ret = dlgitem_set_focus($win,"matchMarkup")
```

dlgitem_set_foreground_at

dlgitem_set_foreground_at(*window*, *listtag*, *row*, *foreground*)

Sets the foreground color for the outline list tag identified by *listtag* in *window*. If *listtag* does not refer to an outline list tag, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

window
the dialog window identifier

listtag
an outline list tag (case sensitive)

row
an offset into listtag(starting from one)

foreground
a named color or an RGB specification preceded with #

dlgitem_set_list_array

dlgitem_set_list_array(*window*, *dlgitem*, *values*)

Sets the entire list of values in the list dialog item or list tag (in outline lists) identified by *dlgitem* in *window*. If *dlgitem* does not refer to a list dialog item or list tag, or if *values* is an associative array, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *values* parameter specifies the non-associative array of values to assign.

Example

```
Values[1] = "Oak"
Values[2] = "Maple"
Values[3] = "Gum"
Values[4] = "Ginkgo"
Values[5] = "Birch"
```

```
Values[6] = "Aspen"
$ret = dlgitem_set_list_array($win, "Foliage", $Values)
```

dlgitem_set_list_at

dlgitem_set_list_at (*window*, *dlgitem*, *index*, *value*)

Assigns a value to the list of values in the list dialog item or list tag (in outline lists) identified by *dlgitem* in *window*. If the list has insufficient entries to accommodate *index*, it is extended with empty strings. If *dlgitem* does not refer to a list dialog item or list tag, or if *index* is illegal, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *index* parameter specifies the index into list on *dlgitem*. The *value* parameter specifies the actual value to assign.

Example

```
$ret = dlgitem_set_list_at($win, "Languages", 1, "C")
$ret = dlgitem_set_list_at($win, "Languages", 2, "C++")
```

dlgitem_set_list_count

dlgitem_set_list_count(*window*, *dlgitem*, *count*)

Sets the number of items in the list associated with *dlgitem* in *window* to *count*. If successful, the function returns a one (1). Returns (-1) if the item does not have an associated list; *dlgitem* may also refer to a list tag (in outline lists).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *count* parameter is an item count

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the LIST_COUNT attribute of the dialog item. The list will be shortened or lengthened as needed.

When used on the root of an outline list item, sets the count of the top-level list tag of the list.

Example

```
$ret = dlgitem_set_list_count($win, "Cars", 3)
```

Sets the number of items in the `Cars` dialog item to 3.

dlgitem_set_listtag_branch_image_at

dlgitem_set_listtag_branch_image_at(*window*, *listtag*, *row*, *image*)

Associates *image* with the outline list entry *row* in the outline list *listtag* in *window*. The *image* appears at the front of the list item. Use this function when you want a specific branch listtag to appear with an image different from the default branch image. If you were to create a file browser using an outline list, you may use a closed folder icon as a default branch image, and a special icon for a locked folder. If *listtag* does not refer to an outline list listtag, `$ERROR` is set and zero (0) is returned. Otherwise, one (`1`) is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list list tag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one).

Example

```
ret = dlgitem_set_listtag_branch_image_at(win, "Hierarchy", 3, "Lockedfolder")
```

This sets the image for the third item in the outline list item `Hierarchy` to the `Lockedfolder` image.

dlgitem_set_listtag_extra_image_at

dlgitem_set_listtag_extra_image_at(*window*, *listtag*, *row*, *image*)

Associates *image* with the outline list entry *row* in the outline list *listtag* in *window*. The image appears at the front of the list item, just to the right of the leaf image set by the `dlgitem_set_default_leaf_image` function. If *listtag* does not refer to an outline list listtag, `$ERROR` is set and zero (0) is returned. Otherwise, one (1) is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list listtag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *image* parameter specifies the ACL Designer image resource to place at the front of the list item.

Example

```
ret = dlgitem_set_listtag_extra_image_at(win, "Hierarchy", 3, "Padlock")
```

sets the image for the third item in the outline list item `Hierarchy` to the `Padlock` image.

dlgitem_set_listtag_leaf_image_at

dlgitem_set_listtag_leaf_image_at(*window*, *listtag*, *row*, *image*)

Associates an ACL Designer image resource *image* with the outline list entry *row* in the outline list *listtag* in *window*. The image appears at the front of the list item. Use this function when you want a specific listtag to appear with an image different from the default image. If you were to create a file browser using an outline list, you may use a file icon as a default leaf image for file leaves, and a special icon for graphic file leaves. If *listtag* does not refer to an outline list listtag, `$ERROR` is set and zero (0) is returned. Otherwise, one (1) is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list listtag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *image* parameter specifies the ACL Designer image resource to place at the front of the list item.

Example

```
ret = dlgitem_set_listtag_leaf_image_at(win, "Hierarchy", 3, "Graphic")
```

sets the image for the third item in the outline list item `Hierarchy` to the `Graphic` image.

dlgitem_set_listtag_openbranch_image_at

dlgitem_set_listtag_openbranch_image_at(*window*, *listtag*, *row*, *image*)

Associates an ACL Designer image resource *image* with the outline list entry *row* in the outline list *listtag* in *window*. The *image* appears at the front of the list item when it is expanded. Use this function when you want a specific expanded branch listtag to appear with an image different from the default image. If you were to create a file browser using an outline list, you may use a open folder icon as a default open branch image and a special icon for open branches with locked contents. If *listtag* does not refer to an outline list listtag, `$ERROR` is set and zero (0) is returned. Otherwise, one (`1`) is returned.

The *window* parameter is a window identifier. The *listtag* parameter is an outline list listtag (case sensitive). The *row* parameter is an offset into *listtag* (starting from one). The *image* parameter specifies the ACL Designer image resource to place at the front of the list item.

Example

```
ret = dlgitem_set_listtag_openbranch_image_at(win, \
 "Hierarchy", 3, "OpenFolderLocked")
```

sets the image for the third item in the outline list item `Hierarchy` to the `OpenFolderLocked` image.

dlgitem_set_mnemonic

dlgitem_set_mnemonic (*win*, *dlgitem*, *mnemonic*)

Sets the mnemonic of the dialog item specified by *dlgitem* in the window specified by *win* to the letter specified by *mnemonic*. If *win* is invalid, or if *dlgitem* does not exist within *win*, 0 is returned. If successful, the function returns 1.

The *win* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *mnemonic* parameter is a letter to set as the mnemonic for the specified dialog item (lower case). In order for the mnemonic to be properly set on the dialog item, the letter *mnemonic* specified must occur in the label of the dialog item. To avoid conflicts with other dialog item mnemonics, you should set each dialog item in a given dialog to a unique mnemonic.

Example

```
$ret = dlgitem_set_mnemonic($win,"matchMarkup","r")
```

dlgitem_set_multiple

dlgitem_set_multiple (*window*, *attribute*, *values*)

Sets the value of the attribute named *attribute* to multiple *dlgitems* within *window*, per the array in *values*. The *values* parameter is an associative array, indexed by the tag names of the dialog items within *window*, containing new values for the attribute. There need not be an entry in *values* for every dialog item in the window. It is an error for *values* to contain a tag name which does not correspond to a *dlgitem*, however. If an error occurs, $ERROR is set and 0 is returned. If successful, the function returns a one (1).

The *window* parameter is a window identifier. The *attribute* parameter specifies the attribute to set. The *values* parameter is an associative array of new values for dialog items.

Notes:

Assignment of values does not cease when an erroneous entry is detected.

Example

```
$Values["NameField"] = "Jeff"
$Values["AgeField"] = "35"
$Values["StateField"] = "MD"
$ret = dlgitem_set_multiple($win, "VALUE", Values)
```

Sets the following values: **NameField** dialog item to "Jeff", **AgeField** dialog item to "35", and **StateField** dialog item to "MD".

dlgitem_set_refresh

dlgitem_set_refresh (*window*, *dlgitem*, *refreshstatus*)

Activates or deactivates the automatic refresh for outline list *dlgitem* in *window*. Turning off the refresh is good if you are executing a routine which would redraw the list several times (for example, fill the list, resort the list, etc.). If *dlgitem* does not refer to an outline list listtag, $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *refreshstatus* parameter sets the refresh/no refresh status of the list entry (1=refresh; 0=no-refresh).

Example

```
$ret = dlgitem_set_refresh($win, "Filelist", 0)
```

Deactivates automatic refresh for the outline list `Filelist`.

dlgitem_set_selection

dlgitem_set_selection(*window*, *dlgitem*, *start*[, *end*])

If *dlgitem* is a outline list item in window, this function selects the entry row that has application data that matches either *start* or *end* (when supplied).

If *dlgitem* is a textbox control (or an editable combobox control) in *window*, `dlgitem_set_selection` highlights a range of text in the text field starting at position *start* and ending at position *end*. If *end* is not supplied, the highlighting continues to the end of the text. If *start* and *end* are the same, no text is highlighted and the cursor is placed at that position.

If `dlgitem_set_selection` is called with any other type of dialog box item, it returns `0` and sets *$ERROR*. Otherwise, the function returns `1`.

dlgitem_set_table_cell_at

dlgitem_set_table_cell_at(*window*, *dlgitem*, *row*, *column*, *text*)

Assigns a value to the cell specified by the intersection of *row* and *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *row* — The 1-based index of the row containing the cell.
- *column* — The 1-based index of the column containing the cell.
- *text* — Specifies the value in the cell.

If *dlgitem* does not refer to a table, or if *row* or *column* are illegal values, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_table_column_align

`dlgitem_set_table_column_align``(`*`window`*`, `*`dlgitem`*`, `*`column`*`, `*`align`*`)`

Sets alignment information for the column specified by *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The 1-based index of the column. If the table has fewer columns than the value of *column*, the table is extended.
- *align* = `LEFT` | `RIGHT` | `CENTER`

  Specifies the alignment information.

If *dlgitem* does not refer to a table, or if *column* or *align* are illegal values, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_table_column_count

`dlgitem_set_table_column_count``(`*`window`*`, `*`dlgitem`*`, `*`count`*`)`

Specifies the number of columns in the table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *count* — Specifies the number of columns in the table.

If *dlgitem* does not refer to a table, or if *count* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_table_column_header_at

`dlgitem_set_table_column_header_at``(`*`window`*`, `*`dlgitem`*`, `*`column`*`, `*`label`*`)`

Assigns a header label to the column specified by *column*.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *column* — The 1-based index of the column. If the table has fewer columns than the value of *column*, the table is extended.
- *label* — The new header label of the column.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_table_row_count

dlgitem_set_table_row_count(*window*, *dlgitem*, *count*)

Specifies the number of rows in the table.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.
- *count* — Specifies the number of rows in the table.

If *dlgitem* does not refer to a table, or if *count* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_table_selection

dlgitem_set_table_selection(*window*, *dlgitem*, *row*[, *column*])

This function selects a row or a cell depending on the selection type of the table control. `dlgitem_set_table_selection` returns `0` if the operation failed. Otherwise, it returns `1`.

- *window* — The window identifier of the window containing the table control.
- *dlgitem* — The value of the table control *id* attribute.
- *row* — The `1`-based index of the row containing the cell.
- *column* — The `1`-based index of the column containing the cell. This value is only used when the table control’s selection type is cell selection.

dlgitem_set_table_sort

dlgitem_set_table_sort(*window*, *dlgitem*, *column*[, *sortorder*])

Specifies on which column the table is to be sorted.

- *window* — The window identifier of the window containing the table.
- *dlgitem* — The value of the control's *id* attribute.

- *column* — The `1`-based index of the column on which the table is to be sorted. If *column* is set to `0`, the table will not be sorted.
- *sortorder* — When `0` or omitted, the column specified in the function will be sorted in ascending order. Otherwise, the column will be sorted in descending order.

If *dlgitem* does not refer to a table, or if *column* is illegal, *$ERROR* is set and `0` is returned. Otherwise, `1` is returned.

dlgitem_set_value

`dlgitem_set_value` (*`window`*, *`dlgitem`*, *`value`*)

Sets the VALUE attribute of the *dlgitem* to *value*. If *window* is invalid, if *dlgitem* does not exist within window, or if *value* does not contain a value of the appropriate type, $ERROR is set and `0` is returned. If successful, the function returns a one (`1`).

- *window* — The window identifier.
- *dlgitem* — The value of the control's *id* attribute.
- *value* — String data for the VALUE attribute.
    - For checkboxes, the value is `0` or `1` for two-state checkboxes and `0`, `1`, or `-1` for tri-state checkboxes. For tri-state checkboxes, `-1` is the indeterminate state.
    - For menuitems and radiobuttons, the value is `0` or `1`. `0` is unchecked and `1` is checked.
    - For comboboxes, listboxes, tablecontrols, radiogroups, and tabboxes, the value is a `1`-based index of the selected item.
    - For descriptions, labels, textboxes, unitdimensionboxes, colordropdowns, sliders, spinners, and datetime, the value is a string.

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the VALUE attribute of the dialog item.

Examples

```
$ret = dlgitem_set_value($win, "TextField", "Machu Pichu")
```

This function can be used to set the active tab of a tabbox control.

```
LINK_NOTEBOOK = "id"
# tab_value is an integer, which represents the 1-based
# position of the tabs by counting from left to right
# within the notebook.
function activate_tab(win, tab_value) {
 dlgitem_set_value(win, LINK_NOTEBOOK, tab_value)
```

```
}
```

dlgitem_show

dlgitem_show (*window*, *dlgitem*)

Ensures that the *dlgitem* is visible when *window* is open.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Notes

Calling this function has the same effect as calling the `dlgitem_set` function on the VISIBLE attribute of the dialog item with a non-zero value.

Any necessary display updates will be scheduled but not necessarily completed when this functions returns.

Example

```
$ret = dlgitem_show($win, "Salary")
```

Shows the `Salary` dialog item.

dlgitem_withdraw

dlgitem_withdraw (*window*, *dlgitem*)

Ensures that the *dlgitem* is invisible when *window* is open. This function is specifically useful on the toolbar, since it automatically invokes a re-display of the toolbar, shifting the other toolbar buttons to the left to fill the space left by the button withdrawn (if the window is already open, you will see a momentary flash as this re-display takes place). If *dlgitem* does not refer to a list dialog item or a list tag (in outline lists), $ERROR is set and 0 is returned. Otherwise, 1 is returned.

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute.

Example

```
$ret = dlgitem_withdraw($win, "Toolbar_Equation")
```

Withdraws the **Equation** toolbar button.

dobj_is_other_locked

dobj_is_other_locked (*dobj*)

Returns a boolean value indicating whether the document object *dobj* is locked by another user. If *dobj* is locked by another user, the function returns 1 (one). If *dobj* is invalid, or is not locked by another user, the function returns 0 (zero).

doc_cache_base

doc_cache_base (*doc*)

Call the `doc_cache_base` function to obtain the base file name that will be used to identify cache files when the document is formatted. If the optional parameter *doc* is omitted, the current document is used.

If this function were to return the path and base file name `/tmp/aptcache/demo000/xyz`, then the `format` command on page 653 will produce formatting files using the base file name `xyz`. For example, formatting files would be named `/tmp/aptcache/demo000/xyz.dvi`, `/tmp/aptcache/demo000/xyz.tex`, and so on.

doc_cache_dir

doc_cache_dir (*doc*[, *create*])

The `doc_cache_dir` function returns the path name of the cache directory associated with the document specified by *doc*. If no cache directory exists, it will be created if the *create* parameter is specified and is non-zero (True). If the *create* is not specified or is zero (False), then it returns the null string if the cache directory has not been created.

The cache directory is used by the formatter to store intermediate files.

doc_clean_cache

doc_clean_cache (*doc*)

Use the `doc_clean_cache` function to remove the auxiliary publishing files stored in the Arbortext Editor cache directory for the document specified by *doc*. The *doc* parameter is the document identifier, as returned by the `current_doc` function. If you invoke this command from the Arbortext Editor command line without the *doc* parameter, `doc_clean_cache` applies to the document in the current Edit window.

Examples:

```
doc_clean_cache()
doc_clean_cache(current_doc())
doc_clean_cache(5)
```

doc_clear

doc_clear (*doc*)

This function removes all the content from the document identified by *doc* leaving an empty document. It does not remove any entity declarations.

The operation cannot be undone.

Note

The `doc_clear` function does not prompt the user about saving changes. This function is primarily intended for manipulating temporary documents not opened by the user.

doc_clone

doc_clone (*doc*[, *flags*[, *viewchgtrk*]])

This function creates an independent copy of the document specified by the *doc* parameter. This function returns a document identifier that may be subsequently called by other functions.

The following flags can be set:

- `0x01` — No content will be cloned. This will result in an empty document.
- `0x02` — Resolve any change tracking markup according to the value of the `set viewchangetracking` option on page 923 for the current view of the specified document. If there is no view setting associated with the specified document, this function uses the global value of `set viewchangetracking`. The `set viewchangetracking` values interacts with this function in the following ways:
  - `original` — The cloned document will have the original markup (changes not applied) but no change tracking markup.
  - `changesapplied` — The cloned document will have the latest markup (changes applied) but no change tracking markup.
  - `changeshighlighted` — The cloned document will have the change tracking markup (no data is lost; changes are still tracked).
- `0x04` — Makes the empty document not inherit entity declarations from the source document. Applied only if `0x01` is also set.
- `0x40` — The cloned document will have the same name as the source document.

Without `0x40` set, the cloned document will have no name. You can use the `doc_dir` function on page 316 to obtain the path to the cloned document.

Note

Avoid using the `doc_set_path` function on page 331 to provide a path identical to the source document, because any subsequent changes made to either document will be reflected in the other document.

The optional *viewchgtrk* parameter can override the current setting of the `set viewchangetracking` option.

doc_close

`doc_close` (*`doc`*)

This function frees all memory used to represent the document tree identified by *doc* and marks the document identifier as invalid. You should not use the document specified by *doc* after you call this function. You cannot use *doc* to specify a command window document; for example, `doc_close(window_doc("cmd"))` is illegal. The function returns `1` on success, `0` on failure.

`doc_close` prompts you to save the document if the document is attached to an edit class window and has been modified. The `doc_close` function returns `0` if the user selects **Cancel** on the **Save** dialog box. To avoid a prompt in this case, a `save` or `set modified=off` command should be done before calling `doc_close`.

If *doc* specifies a document in a window not created by the `window_create` function on page 588, for example an auxiliary help window created by the `help` command on page 655, `doc_close` also removes the window. For example, `doc_close(window_doc("helpwin2"))` would free the document tree representing the contents of the `helpwin2` window, and destroy the `helpwin2` window.

doc_compose_stylesheets

`doc_compose_stylesheets`(*`[doc]`*)

The `doc_compose_stylesheets` function returns `1` if publish dialog boxes for the document type associated with *doc* allow you to specify a stylesheet. The function returns a `0` if you can't specify stylesheets in the publish dialog boxes.

If *doc* is omitted or `0`, the current document is used.

doc_default_stylesheet_path

doc_default_stylesheet_path([*use*[, *doc*]])

This function returns the full path and file name for the stylesheet that will be used by default for the given *use* and document *doc*. Valid values for *use* are:

- 0 — Arbortext Editor
- 1 — Print and PDF
- 2 — HTML File
- 3 — HTML Help
- 4 — Web
- 5 — RTF

If no document is specified, the *doc* parameter is the value of current_doc.

If, for example, you opened a document with the doc_open() function, and passed the 0x200 flag to not load a stylesheet, you could then use doc_default_stylesheet_path(0, doc) to determine which stylesheet would be used as the editing stylesheet if you were to pass the document to doc_show().

The value returned does not reflect what stylesheets the document is currently set to use. It reflects what stylesheets would be used for the document if the document were opened in Arbortext Editor with no special application code. If *use* is 0 (Arbortext Editor), the stylesheet returned is determined by stylesheet association if one exists, or name and location. For example, by docname.style or doctype.style if such a file is found. For other uses, an empty string is returned unless there is a stylesheet association for the specified use.

doc_delete_stylesheet_association

doc_delete_stylesheet_association(*doc*, *i*)

This function deletes the i^{th} stylesheet association for document *doc*.

If *doc* is not a valid document identifier, or if there is no i^{th} stylesheet association for *doc*, the function returns 0. Otherwise, it returns 1.

doc_dir

doc_dir(*doc*)

This function returns the directory of the document specified by the *doc* parameter. If the optional *doc* parameter is omitted, the current document is assumed.

The doc_dir function can retrieve the document path to a cloned document using the document identifier returned by the doc_clone function on page 314.

doc_dtd_not_defined

doc_dtd_not_defined ([*doc*])

This function can be used to distinguish between the two instances in which the function doc_freeform returns 1 (True). If the document identified by *doc* is an XML instance without a DTD declaration, doc_dtd_not_defined returns 1 (True). If the document's declared DTD cannot be found, and the user has selected to edit in tagged mode without a DTD, this function will return 0.

If doc_freeform returns 0, doc_dtd_not_defined will also return 0.

The *doc* argument specifies the identifier of the document to query. If omitted or 0, the current document is used.

doc_dtd_not_found

doc_dtd_not_found ([*doc*])

This function can be used to distinguish between the two instances in which the function doc_freeform returns 1 (True). If the document identified by *doc* is an XML instance with a DTD declaration, but the DTD cannot be found, doc_dtd_not_found returns 1 (True).

If doc_freeform returns 0, doc_dtd_not_found will also return 0.

The *doc* argument specifies the identifier of the document to query. If omitted or 0, the current document is used.

doc_estimate_dfs

doc_estimate_dfs ([*doc*])

This function returns an estimate of the number of document fragments in a document.

Taking into consideration the overall size of the document, this value can help determine the setting for the set bigjobthreshold option on page 758. The doc__estimate_dfs function is also used when publishing to determine if content pipeline output should be written to disk rather than to memory. Refer to the description of the set bigjobthreshold option on page 758 for details.

For more information on using Arbortext Publishing Engine for publishing, refer to the *Programmer's Guide to Arbortext Publishing Engine*, the *Programmer's Reference*, and the *Content Pipeline Guide*.

doc_first_dobj

doc_first_dobj (*doc*)

Returns the first document object in the given document *doc*. If *doc* is not specified, the current document is used.

doc_flatten

doc_flatten (*type*, *delete_declarations*, *doc*)

This function flattens a file; substituting the contents of an entity reference with the actual content of the entity. You cannot undo this operation.

> **Note**
>
> The "Entity XXX referenced in another entity" warnings result from recursively defined entities and can usually be ignored.

The `type` string specifies:

- `file` — flattens file entities only.
- `text` — flattens text entities only.
- `both` — flattens both text and file entities.
- `include` — flattens XIncludes only.
- `all` — flattens all text and file entities as well as XIncludes.
- `conref` — flattens DITA content references only.

You may specify more than one type by connecting the types with a plus sign (+). For example, the following value for `type` would flatten both XIncludes and DITA content references: `include+conref`.

delete_declaration is an optional boolean parameter that controls if the entity declarations should be removed from the document. `1` means remove the entity declarations, `0` means keep the entity declarations. The default is to keep the declarations. For example, if *type* is `all`, and *delete_declaration* is `1`, then all file and text entity declarations will be deleted.

doc is an optional document identifier. The default value is the current document.

doc_format_needed

doc_format_needed ([*doc*[, *needed*]])

This function returns and optionally sets the format needed state of an open document as displayed in the status bar. If the *needed* parameter is specified, the state is set according to the following values:

- `0` — Clears the format-needed status.
- `1` — Sets the format-needed status.
- `2` — Sets the reformat-needed status (when another formatting pass is required).

`doc_format_needed` returns the previous state. If the *doc* parameter is omitted or `0`, the current document is used.

doc_formattable

`doc_formattable` `([`*`doc`*`])`

This function returns 1 (true) if the document specified by *doc* can be formatted for published output. It returns 0 for ASCII documents or FOSIs. In Arbortext Editor, it always returns 0.

If *doc* is omitted or 0, the current document is used.

doc_freeform

`doc_freeform` `([`*`doc`*`])`

This function checks whether a document is being edited in tagged mode without a DTD. This function returns `1` (True) if the document identified by *doc* is an XML instance without a DTD declaration. This function also returns `1` if the document's declared DTD declaration cannot be found and the user selected to edit in tagged mode without a DTD. For all other documents, this function returns `0`.

The *doc* argument specifies the identifier of the document to query. If omitted or `0`, the current document is used.

doc_from_compare

`doc_from_compare` `([`*`doc`*`])`

This function determines whether a specified document was generated by a document comparison operation. The *doc* parameter is a document identifier.

The return value is 1 if the document was generated by a compare operation. The return value is 0 if it was not.

doc_get

doc_get(*doc*, *attr*)

This function returns the document scope value of the set option *attr* for the document identified by *doc*. If *attr* is a `set` option with a local scope other than document, or no local scope at all, it returns nothing.

Examples

```
doc_get(doc, "showiconsfulltags")
doc_get(doc, "textentityfontcolor")
```

doc_get_stylesheet_association

doc_get_stylesheet_association(*doc*, *i*, *params[]*)

This function gets information about a stylesheet association. It returns data for the i^{th} stylesheet association of document *doc* in associative array *params* where the keys of the array elements are `href`, `type`, `title`, `media`, `charset`, and `alternate`, corresponding to the attributes of the stylesheet processing instruction. The `href` element will contain a relative path name, an absolute path name, or an `http://URL`.

For example, the following ACL function would display the information for each existing stylesheet association for a document:

```
function show_ss_associations(doc = current_doc())
{
 local n = doc_num_stylesheet_associations(doc);
 if (n < 0)
 {
 response("Bad doc $doc passed to show_ss_associations from " . caller(1));
 return 0;
}
 local i;
 eval "Stylesheet associations:" output=*
 for (i=1; i <= n; ++i)
 {
 local params[];
 if (!doc_get_stylesheet_association(doc, i, params))
 {
 response("Bad call to doc_get_stylesheet_association in " . caller());
 return 0;
}
 eval "href =", params["href"], " type =", params["type"], \
 " title =", params["title"], \
 " media =", params["media"], " character set =", params["charset"], \
 " alternate =", params["alternate"] output=>*
}
 return 1;
}
```

If *doc* is not a valid document identifier, or if there is no i^{th} stylesheet association for *doc*, the function returns 0. Otherwise, it returns 1.

doc_get_translation_locale

doc_get_translation_locale([*doc*])

Returns the translation locale of a specific document if the document is a translated version of another document. Otherwise, `doc_get_translation_locale` returns the empty string.

- *doc* — (Optional). The identifier of the document for which the translation locale is returned. If *doc* is omitted or `0`, the current document is used.

doc_get_translation_orig_uri

doc_get_translation_orig_uri([*doc*])

Returns the original URI value (if known) of the document for which a specific document is a translation. Otherwise, `doc_get_translation_orig_uri` returns the empty string.

- *doc* — (Optional). The identifier of the document for which the URI is returned. If *doc* is omitted or `0`, the current document is used.

doc_has_change_tracking

`ret` = **doc_has_change_tracking**([*doc*])

This function returns `1` only if change records are found in *doc*. If *doc* is not specified, the value of `current_doc` is used.

doc_incomplete

doc_incomplete ([*doc*[, *flags*]])

This function returns `1` (true) if the document specified by *doc* is currently marked incomplete. If the *flags* parameter is specified and non-zero, then a `check_completeness` command is run first to update the incomplete status.

Note

Normally, once a document is marked incomplete, the state is not reset until a completeness check is requested.

The *flags* parameter is a bitmask that controls the options passed to the `check_completeness` command. It is constructed by using OR for the flags from the following list:

- 1 — run the `check_completeness` command

  If this value is used alone, the function checks the document for completeness and sets its completeness flag to 0 or 1 accordingly.

  If this value is not specified, but 2, 4, or 8 are, it is assumed that 1 was also specified.
- 2 — do not insert missing required elements
- 4 — generate error messages from the `check_completeness` command, including any parser error messages caused by the loading of required file entities, and display them in dialog windows as needed
- 8 — generate error messages from the `check_completeness` command, but do not display them in a window

  With this flag set, a completeness check will complete without stopping to wait for a user to acknowledge message windows.

  Each message generated is offered to ACL code via the context error hook. A batch ACL script can use `doc_incomplete(doc, 8)` to execute a full completeness check and obtain errors without any dialogs being displayed.

  If both 4 and 8 are set, 8 is ignored.

Note the following when working with `doc_incomplete()`:

- If *flags* is omitted or `0`, the `check_completeness` command is not run.
- If *doc* is omitted or `0`, the current document is used.
- Running `doc_incomplete()` will only include checks for empty elements if this combination of settings exists:
  - The preference `emptyelementswarnings` is set to `on` (this is not the default value)
  - `doc_incomplete()` specifies flag 4 or flag 8

doc_invalidate_graphics

`doc_invalidate_graphics``([`*`doc`*`])`

This function resolves, reloads, and redraws (if displayed) all graphics in the document *doc*.

If *doc* is omitted or `0`, the current document is used.

doc_is_translation

doc_is_translation ([*doc*])

Determines whether a specific document is a translated version of another document, based on whether the Translation PI is present in the document.

- *doc* — (Optional). The identifier of the document for which the translation status is checked. If *doc* is omitted or `0`, the current document is used.

doc_kind

doc_kind ([*doc*])

This function returns a string indicating what "kind" of document is specified by *doc*. If the *doc* parameter is omitted, the current document is assumed. The possible return values are:

- `sgml` — indicates an SGML document.
- `xml` — indicates an XML document.
- `html` — indicates an HTML document.
- `ascii` — indicates an untagged ASCII document.

doc_list

doc_list (*arr*[, *flags*])

This function fills the array *arr* with a list of all existing documents, returning the number of document identifiers stored in the array.

flags is a bitmask that specifies which document identifiers are returned. It is constructed by ORing the flags from the following list:

- `1` — include command window documents
- `2` — include paste buffers
- `4` — include "textwin" documents, that is, documents associated with the built-in text windows `helpwin[1-4]` and `msgwin[1-4]`.
- `16` — include file entity documents
- `32` — include FOSI documents
- `64` — include top level documents that use a DTD and are not file entities or paste buffers.
- `128` — include documents that use a built-in document type, such as `ascii` or `help`, which is not matched by any of the flags specified above (that is, does not include documents matched by bits 1, 2, 4, and 8).

If *flags* is omitted or `0`, then all documents are returned as if all bits were specified.

doc_name

`doc_name` (*`doc`*)

This function returns the name associated with the document given by *doc*, or the null string if the document was created without a name argument specified.

doc_namecase_sensitive

`doc_namecase_sensitive`(*`doc`*)

This function determines if the given document is case sensitive when dealing with markup. The function returns a one (`1`) or zero (`0`) based on the `NAMECASE GENERAL` setting of the document type definition (DTD) associated with the optional document ID parameter *doc*. If *doc* is omitted, the current document is used. A return of zero (`0`) means case-insensitive (that is, `NAMECASE GENERAL YES`). A return of one (`1`) means case-sensitive (that is, `NAMECASE GENERAL NO`). This function always returns one (`1`), if *doc* is an XML document, because XML markup is always case-sensitive.

doc_new_stylesheet_association

`doc_new_stylesheet_association`(*`doc`*, *`params[]`*)

This function creates a new stylesheet association for document *doc* with the attribute values specified by the associative array *params* where the keys of the array elements indicate the attribute name. Valid keys are `href`, `type`, `title`, `media`, `charset`, `alternate`. There must be elements for `href` and `type`. Others can be omitted. The `href` element should contain a relative path name, an absolute path name, or an `http://URL`. Arbortext Editor processes `media` values of Editor, Print/PDF, and screen. The screen media type is used to designate the HTML file stylesheet.

For example, the following function creates a new stylesheet association:

```
function add_ss_association(doc = current_doc())
 {
 local params[];
 params["href"] = "htmlhelp.xsl";
 params["type"] = "text/xsl";
 if (!doc_new_stylesheet_association(doc, params))
 {
 response("Bad call to doc_new_stylesheet_association in " . caller());
 return 0;
```

```
}
 return 1;
}
```

If *doc* is not valid, or if *params* does not include elements for `href` and `type`, the function returns `0`. Otherwise, it returns the resulting number of stylesheet associations.

If *params* includes elements with keys other than those listed above, those elements are silently ignored. There is no validation of the contents of any of the *params* elements.

doc_next_dobj

doc_next_dobj (*dobj*)

Returns the document object that follows *dobj* in the same document.

doc_num_stylesheet_associations

doc_num_stylesheet_associations(*doc*)

This function returns the number of stylesheet associations that are stored in the document. If *doc* is not a valid document identifier, the function returns `-1`.

doc_open

doc_open ([*path*[, *flags*[, *name*[, *pubid*[, *sysid*[, *stylesheet*]]]]]])

This function creates a new document tree and returns an identifier that may be used in subsequent calls to `current_doc` on page 255, `oid_caret` on page 430, `insert_tag` on page 387, and other functions. This function also generates an initial context for fragments so that context rules can be turned on for a partial document when displayed in a window using the `doc_show` on page 332 command. The arguments that follow are optional.

path — Specifies the path name of a document directory or file name to load the initial contents of the document tree. If not specified or a null string, the document tree is empty.

flags — A bitmask that specifies open options and is constructed using `OR` with the following flags:

- `0x001` — 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 checkout status of CMS objects will not be affected.

- `0x002` — 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 checkout status of CMS objects will not be affected.

- `0x004` — Do not lock the underlying file. Overrides all other flags which might acquire a file lock. The resulting document will not be modifiable unless `0x002` is also given.

  The checkout status of CMS objects will not be affected.

- `0x008` — Do a completeness check when reading the SGML or XML file. This corresponds to the `-cc` option for startup and the `edit` command.
- `0x010` — Do not do a completeness check when reading the SGML or XML file. This corresponds to the `-nocc` option for startup and the `edit` command. By default, a completeness check is not done if the file was saved by Arbortext Editor. This bit is ignored if bit 4 is also specified.
- `0x020` — Do not display any parser error messages in a message window. Instead, ignore all warnings and errors.
- `0x040` — The document type specified by *pubid* and *sysid* should be used to parse the SGML or XML file instead of the document type specified in the file itself.

  If the *pubid* and *sysid* refer to a schema (not a DTD), the `0x40000` bit must also be set.

- `0x080` — Open a help document. (Used internally by Arbortext Editor.)
- `0x100` — Open a new document as an XML document when a public or system ID is specified.
- `0x200` — Open the document without loading a stylesheet. If the document is called later by `doc_show`, the required stylesheet is loaded at that time.

> **Note**
>
> When an application opens a document without a stylesheet using this flag, a subsequent write or save operation will not load the stylesheet. Without a stylesheet, all original line breaks in the document are preserved during the write or save. If you do not want to preserve the original line breaks, do not specify this flag when opening a document.

- `0x400` — Do not prompt the user if the document type associated with the document does not exist or is not compiled. Instead, return `-1`.
- `0x8000` — Source the associated document type instance files (`instance.acl` and `instance.js`), the document command files (`docname.acl` and `docname.js`), and any applications in the `custom` directory's `editinit` subdirectory. By default, these files are not processed until the document is displayed in a window with the `doc_show` function or `edit` command. The editfilehook on page 946 is not called by `doc_open` but only if `doc_show` or `edit` is used.
- `0x10000` — Treat the document as if it were created using **File ▸ New**. In this case, the path name is set to null and the document name is of the form `Document`*n*.
- `0x20000` — Specifies that if an autosave or recovery file exists for the document, the user should be prompted to select the document to open.
- `0x40000` — Specifies that the *pubid* parameter is actually a namespace URI instead of a public identifier. If the `0x040` bit is also specified, then the namespace URI is used to locate the XML schema to parse the document. The *sysid*, if given, specifies the path to the DTD/schema file and is used if the namespace URI is null or is not resolved by catalog lookup.
- `0x80000` — Open the document in free form mode, ignoring the document type specified in the file or by the public identifier *pubid* and system identifier *sysid* parameters. Flag bit `0x100` (open as XML) is implied by this bit.
- `0x200000` — Specifies that the path name parameter *path* is actually a string to parse instead of a file to open. If the string does not contain a DOCTYPE declaration, the *pubid* and or *sysid* parameters must be given so the desired document type is used to parse the string. Otherwise, the `0x80000` bit must be specified. If the string contains XML markup but does not start with an XML declaration, the `0x100` bit must also be specified.
- `0x1000000` — Do not add the document to the **File** menu's list of most recently used documents if this document is subsequently opened in an Edit view.

name — Specifies a name to be used for informational purposes. If omitted or null, the base name of *path* if given is used. If *path* is omitted, an internal name is assigned.

pubid — Specifies the public identifier for the document type.

sysid — Specifies the system identifier of the document type.

The *pubid* and *sysid* arguments specify the document type for the document tree if *path* is omitted or null or if the associated SGML file does not specify a DOCTYPE declaration, or if bit `0x040` is included in *flags*. The *pubid* and *sysid*

arguments are ignored if *path* specifies an SGML file that starts with a DOCTYPE declaration or a binary document file, unless bit 0x040 is included in *flags*. As a special case, if *pubid* is "help", the built-in help window document type is used. If the document type is not specified, is “ascii”, or cannot be determined, then the `ascii` document type is used.

stylesheet — Specifies the stylesheet to be used instead of the default stylesheet for the document. If bit `0x200` is included in the *flags* parameter, the *stylesheet* parameter is ignored. If the specified stylesheet does not exist, Arbortext Editor displays an error message and the function returns a -1.

The document identifier returned should be deleted with the `doc_close` function when it is no longer needed to unload the document tree from memory.

doc_parent

`doc_parent` (*`doc`*)

This function returns the identifier of the document that references the file entity document given by *doc*, or `-1` if *doc* is not a file entity.

doc_path

`doc_path` (*`doc`*)

This function returns the path name associated with the document given by *doc*, or the null string if no path name was specified when the document was created.

doc_read_only

`ret=`**`doc_read_only`** (`[`*`docid`*`[,` *`bool`*`]]`)

This function returns (and optionally sets) the read-only state specified when opening a document. `doc_read_only` does not report the file permission setting of a file as defined by the operating system, but the value used to open the document. (Documents can be opened as read-only with **Open as read-only** checked on the **Open** dialog box and with the function call `open(doc, "r")`. Documents can also be set to read-only using the function call `doc_read_only(doc,1)`.)

If no parameters are included, `doc_read_only` returns a boolean value (`1` = read-only, `0` = read/write) for the read-only state of the current document. If the *docid* parameter is provided, `doc_read_only` returns the read-only state of the specified *docid*.

If both a *docid* parameter and a *bool* parameter are provided, the function returns the read-only state of the specified *docid*, and then sets the read-only state of the specified document to the boolean value specified in *bool* (`1` = read-only, `0` = read/write).

Note that a standard opening of a read-only file will not cause `doc_read_only(doc)` to return `1`. This is due to the fact that a read-only file on disk may have references (such as XML inclusions and file entities) to other files that are not-read only. `doc_read_only(doc)` returns `0` for such a read-only root document because there may be parts of the document that can be edited.

Examples:

```
ret=doc_read_only()
ret=doc_read_only(docid)
ret=doc_read_only(docid,1)
```

To determine if a given oid can be modified, refer to `oid_protected()` on page 455 and `oid_find_valid_insert()` on page 441.

doc_revert

`doc_revert` `([`*`doc`*`[, `*`flags`*`]])`

This function reverts the document specified by the *doc* parameter to the last saved version, discarding any changes. If *doc* is `0` or omitted, the current document is used.

The *flags* parameter is a bitmask that specifies revert options and is constructed using `OR` with the following flags:

- `0x001` — Revert the document read only. The in-memory document may not be changed in this case.
- `0x002` — Suppress the prompt that unsaved changes will be discarded.
- `0x004` — Revert the XML or SGML document in untagged mode. That is, do not interpret any markup.
- `0x008` — Revert the document using an XML parser.

doc_save

`doc_save` `([`*`doc`*`[, `*`flags`*`[, `*`path`*`]]])`

This function saves a document. If the *doc* parameter is `0`, the current document is saved. If both the *doc* and *flags* parameter is omitted, the current document is saved. If the *flags* parameter is omitted, no flags are set. If the *path* parameter is supplied, the document is saved to the file path set in that parameter instead of the current document. A file name must be supplied in this case if saving an XML document as SGML (or vice-versa).

The following flags are allowed:

- `0x0001` — For documents with change tracking markup, save as if all changes are rejected (original view).
- `0x0002` — For documents with change tracking markup, save as if all changes are accepted (latest view).
- `0x0004` — For documents with change tracking markup, save as if all changes are pending (highlighted view).
- `0x0008` — Write the document as an SGML document.
- `0x0010` — Write a text-only version of the document.
- `0x0020` — Write the document as an XML document.
- `0x0040` — Removes the DOCTYPE header and internal subset, including any private ENTITY declarations.
- `0x0080` — Removes processing instructions specific to Arbortext Editor.
- `0x0100` — Enables entity output conversion.
- `0x0200` — Suppresses entity output conversion.
- `0x0400` — Writes non-ASCII characters as character entity references.
- `0x0800` — Writes non-ASCII characters as characters in the target encoding.
- `0x1000` — Writes non-ASCII characters as numeric character references.
- `0x2000` — Used internally for HTML output.
- `0x4000` — Expands all file entities recursively.
- `0x8000` — Expands all text entities recursively.
- `0x10000` — Writes a non-fragment header, if possible.
- `0x20000` — Expands all XInclude references recursively.

doc_set

`$ret=`**`doc_set`** `(`*`doc`*`, `*`option`*`, `*`value`*`)`

For the document with a doc id of *doc*, this function sets the local document scope for the specified *option* to *value*. If you attempt to set an option with no local scope, or a local scope of window, an error occurs. If you set an option with a view scope, all views for the specified document will be updated to reflect the change in option value.

If the function executes successfully, it returns a one (`1`). If the specified *option* is invalid, or does not have a local scope of document or view, or the value is invalid for the option, the function returns a zero (`0`).

Examples

```
$ret = doc_set($doc,"showiconsfulltags","on")
$ret = doc_set($doc,"tagdisplay","full")
```

doc_set_path

doc_set_path (*doc*, *path*[, *flags*])

This function changes the path name associated with the document given by *doc* to *path*. The new path name will be used to save the document. If *path* is a null string, the document will no longer have an associated path name. In this case, `save_as` must be used to save the document. If the second bit of the optional *flags* parameter is set, then the document will continue to use the same cache directory.

doc_set_stylesheet_association

doc_set_stylesheet_association(*doc*, *i*, *params[]*)

This function sets the *i*th stylesheet association for document *doc* to have the attribute values specified by the associative array *params* where the keys of the array elements indicate the attribute name. Valid keys are `href`, `type`, `title`, `media`, `charset`, and `alternate`. The `href` element should contain a relative path name, an absolute path name, or an http:// URL. Arbortext Editor processes `media` values of `Editor`, `Print`, `PDF`, and `screen`. The screen media type is used to designate the HTML file stylesheet.

For example, this function adds screen as a `media` type to the *i*th stylesheet association without changing the other fields.

```
function add_screen_media(i, doc = current_doc())
 {
 local params[];
 if (!doc_get_stylesheet_association(doc, i, params))
 {
 response("Bad call to doc_get_stylesheet_association in " . caller());
 return 0;
}
 if (params["media"] != "")
 {
 params["media"] .= ",";
}
 params["media"] .= "screen";
 if (!doc_set_stylesheet_association(doc, i, params))
 {
 response("Bad call to doc_set_stylesheet_association in " . caller());
 return 0;
}
 return 1;
}
```

If *doc* is not a valid document identifier, or if there is no i^{th} stylesheet association for *doc*, the function returns 0. Otherwise, it returns 1.

If *params* includes elements with keys other than those listed above, those elements are silently ignored. There is no validation of the contents of any of the params elements.

doc_set_translation_locale

doc_set_translation_locale(*locale*[, *doc*])

Sets the translation locale of a specific document and marks the document as a translated version of another document.

- *locale* — The translation locale to which the document will be set.
- *doc* — (Optional). The identifier of the document for which the translation locale is set. If *doc* is omitted or `0`, the current document is used.

doc_set_translation_orig_uri

doc_set_translation_orig_uri(*uri*[, *doc*])

Sets the original URI value of the document for which a specific document is a translation and marks the document as a translation.

- *uri* — The URI of the document for which *doc* is a translation.
- *doc* — (Optional). The identifier of the document for which the URI is returned. If *doc* is omitted or `0`, the current document is used.

doc_show

doc_show (*doc*, *window*)

This function shows the document tree identified by *doc* in the window specified by *window*, which is a window identifier as returned by `window_create` or is one of the predefined window names `help`, `helpwin[1-4]`, `msg`, or `msgwin[1-4]`. In the latter case, the window need not already exist. In both cases, the new document replaces whatever was previously displayed in the window. The specified document must not already be displayed in another window. `doc_show` returns `1` on success, or `0` if the window is not a supported window for `doc_show`.

If *window* specifies an edit class window, `doc_show` does the same initialization as the `edit` command (that is, document type instance files on page 48 (`instance.acl` and `instance.js`) and document command files on page 48 (*docname*`.acl` and *docname*`.js`) are read if they exist, and `editfilehook` on page 946 is called). Context rules are also enabled for the document.

doc_type

doc_type ([*doc*])

This function returns the external name of the document type associated with the document; for example, docbook. If *doc* is omitted, the current document is used.

If the given document is an XML instance without a DTD declaration, this function returns the name of the first non-empty tag in the document.

doc_type_dcf_file

doc_type_dcf_file ([*doc*])

This function returns the full path and file name of the document type configuration file (`.dcf`) for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

doc_type_description

doc_type_description ([*doc*])

This function returns the first document type description declared in the document type configuration file (`.dcf`) for the document type associated with *doc*. This value is displayed in the **New Document** dialog box. If *doc* is omitted or `0`, the current document is used.

doc_type_dir

doc_type_dir ([*doc*])

This function returns the directory name for the document type associated with *doc*. If *doc* is omitted, the current document is used. The document type directory contains the `.dtd`, `.dcf`, and `.fos` files that describe the document type.

doc_type_dita

doc_type_dita ([*doc*])

This function determines whether a given document *doc* is a DITA document. If *doc* is omitted or set to zero, the current document is used. The function returns one of the following values:

- `0` — The document is not a DITA document.
- `1` — The document is a DITA topic type or a DITA Ditabase.
- `2` — The document is a DITA map type.

doc_type_extension

doc_type_extension ([*doc*])

This function returns the default extension declared in the document type configuration file (`.dcf`) for the document type associated with *doc*. The default extension is used when saving a new document of the document type. If *doc* is omitted or `0`, the current document is used.

doc_type_file

doc_type_file ([*doc*])

This function returns the path name of the DTD (`.dtd`) or XML schema (`.xsd`) that defines the document type associated with *doc*. If *doc* is omitted, the current document is used.

doc_type_gi

doc_type_gi ([*doc*])

This function returns the generic identifier name from the `DOCTYPE` declaration for the document given by *doc*. If *doc* is omitted or 0, the current document is used.

Note that the return value is normally the root element declared in the DTD (for example, “book”), but it may be a different element name if the file had a `DOCTYPE` line specifying a lower level element (for example, “chapter”). This function is different from the `doc_type` on page 333 function that returns the external name of the document type which need not be an element name.

doc_type_language

doc_type_language ([*doc*])

This function returns the string `"XML Schema"` if the document type associated with the specified document was defined using an XML schema, `"DTD"` if by a DTD, or the null string for free form or ASCII documents. If *doc* is not specified or is `0`, the current document is used.

doc_type_namespace

doc_type_namespace ([*doc*])

This function returns the target namespace URI declared in the XML schema for the document type associated with *doc*. If the document type is defined by a DTD but was located using a namespace URI catalog entry, then that namespace URI is returned. Otherwise, a null string is returned.

If *doc* is omitted, the current document is used.

doc_type_namespaces

doc_type_namespaces (*doc*, *arr*)

This function populates an associative array with all of the namespaces imported into the XML Schema for the document type associated with *doc*. The associative array has the namespace prefix as the key, and the namespace URI as the value. The function returns the number of entries in the array.

If *doc* is omitted or `0`, then the current document is used.

doc_type_root_tags

doc_type_root_tags (*doc*, *arr*)

This function fills the array *arr* with the names of all elements which are considered root tags for the document type associated with the document given by *doc*. Any element declared in the XML DTD or XML Schema that is not included in the content model of another element is considered a possible root element.

For compiled document types, only the top tag specified when the document type was compiled is returned.

The function returns the number of elements stored in the array.

doc_type_schematron_file

doc_type_schematron_file([*doc*[, *newFile*]])

This function retrieves or sets the list of Schematron files used with the document type associated with *doc*. If *doc* is not supplied or is `0`, then the current document is used. If *newFile* is not supplied, then the function returns the full paths to the document type's Schematron files. If *newFile* is supplied, then the specified Schematron file is added to the list used with the document type. If *newFile* is supplied and is an empty string, then no Schematron processing will be done for the document type.

Any Schematron file associated with a document type is used by completeness checking for further validations of a document instance.

For example, the following ACL function call will modify the document type schematron file association for the current document to include the files `axdocbook1.sch` and `axdocbook2.sch`:

```
doc_type_schematron_file(0, "axdocbook1.sch;"axdocbook2.sch");
```

Note that the semi colon (`;`) is the required delimiter.

doc_type_xml

`ret = **doc_type_xml** ([*doc*])`

This function returns `1` if the document type associated with the specified document was compiled as an XML application or `0` otherwise. If *doc* is not specified, the value of `current_doc` is used.

doc_update_display

`doc_update_display` `([*doc*[, *flag*]])`

This function suspends screen updates for the specified document *doc* if the *flag* parameter is zero (`0`). If the *flag* parameter is not set, then the function returns the current setting for whether screen updates are suspended for *doc* (`1` if the display is not suspended).

Suspending screen updates may speed up a script that makes many edits to a document that is displayed in a window, especially if partial generated text updates are enabled (the default). Before returning, a script that suspended screen updates would need to re-enable screen updates by calling `doc_update_display` with the *flag* parameter set to `1`.

Use caution when calling this function. If screen updates are not restored, then the document may be displayed incorrectly to the user.

doc_valid

`doc_valid` `(*doc*)`

This function returns `1` (True) if *doc* is a valid document identifier, otherwise it returns `0`.

`doc_valid(0)` is always a valid document and is the same as `doc_valid(current_doc())`.

doc_window

`doc_window` `(*doc*)`

This function returns the identifier of the window associated with the document given by *doc*. If *doc* is invalid or is not currently attached to a window, it returns `-1`.

doc_word_count

doc_word_count ([*doc*[, *flags*]])

Returns the number of words in the current selection or whole document if no selection has been made. The message "*nnn* words in the current selection" is written to the status bar if not suppressed.

- If the *doc* value is omitted, the default is the current doc.
- *flags* — Flag bits ORed together, with a default of 0. Initially, only bit 1 is supported. If set, the message is shown, else it is suppressed.

document_export

document_export ([*inFile*[, *outFile*[, *styleFile*[, *logFile*[, *nFlags*]]]]])

This function starts an export process with the specified parameters. This function returns a one (`1`) if the export was successful. `document_export` returns zero (`0`) if unsuccessful.

`document_export` has the following parameters:

- *inFile* is the path name of the document to export. If null, the current document is used.
- *outFile* is the path name of the converted document.
- *styleFile* is the Arbortext Styler-created style sheet to use during the export.
- *logFile* is the path name of a log file used to record information about the export process.
- *nFlags* is a set of bit flags (that is, an integer argument, not a string argument), defined as follows:
  - If bit `0`,*flagLightsOut* is `1` (decimal value of `1`), the export process is run “lights-out” or batch mode. In this mode, the export process will not prompt the user for anything on screen. The default value for this option is dependent on how Arbortext Editor is being run. The default is `0`.
  - If bit `4`, *flagViewRTFResult*, is `1` (decimal value of `8`), the system-defined

RTF application (typically Microsoft Word) will display the exported RTF file automatically when the process is complete.

Note

The Arbortext Import/Export bit flags are defined as the following ACL constants:

```
################################
# Import/Export bit flags
################################
global flagLightsOut = 0x0001
global flagViewRTFResult = 0x0010
```

The first three parameters (*inFile*, *outFile*, and *styleFile*) are optional in that they need not be specified in the function call. However, if any one of them is not specified, a dialog box will be displayed to allow the user to specify valid values for them. The final two parameters (*logFile*, *nFlags*) are optional. If they are not specified, default values will be used. If the last parameter, *nFlags*, is set to `1`, the first three parameters must be specified because no dialog box will be displayed to allow the user to specify valid values for them. If *nFlags* is set to `1` and one of the first three parameters is not specified, the function will return `0` stating failure of the export process.

dom_address

`addr = dom_address (domId[, interface])`

Returns the address of the DOM object identified by *domId*. The returned address may be passed to a dynamically-loaded entry point with the ACL function `dl_call`.

The optional *interface* parameter specifies the Application Programming Interface that the DOM object will support. If omitted, it defaults to `dom L1 V1.0`, meaning Version 1.0 of Level 1 of the W3C Document Object Model.

dom_document

`domId = dom_document ([doc[, interface]])`

Creates a new DOM document object for the Arbortext Editor document *doc* and returns the DOM document object's ID. If *doc* is omitted, the function uses the returned value of `current_doc`.

The optional *interface* parameter specifies the Application Programming Interface that the DOM object will support. If omitted, it defaults to `dom L1 V1.0`, meaning Version 1.0 of Level 1 of the W3C Document Object Model.

If an error occurs, this function returns the value *h::NullDOMId*.

After creating a DOM document object with `dom_document` and finding the object's address with `dom_address`, the ACL programmer can pass the address to an external dynamically-loaded C++ program using `dl_call`.

dom_free

`ret = **dom_free** (domId)`

Frees the DOM object identified by *domId*. Returns *0* if *domId* is invalid, *1* otherwise.

If the address of a DOM object is passed to an external C++ program, calling `dom_free` will make the address invalid. Any attempt by the C++ program to call the DOM object from that point forward may cause Arbortext Editor to fail.

dom_oid

`domid = **dom_oid** ([*oid*])`

Returns the ID of the DOM object associated with the object ID *oid*. If *oid* is omitted, the oid containing the current caret position is used.

If the document containing *oid* does not have an associated DOM document object, a new DOM document object is created and associated with the document; in this case, the default DOM interface code for the document will be set to `dom L1 V1.0`.

drag_start

`**drag_start** ([*cut*])`

This function turns on drag-and-drop editing. The function tracks the mouse movement and gives the user visual feedback on whether or not the selected area can be pasted at the cursor location. The function also tracks the state of the **ctrl** key to determine if the user intends to copy the selection (**ctrl** key selected during the drag) or cut the selection (**ctrl** key not selected during the drag). The cursor moves with the mouse pointer and indicates where the drop should occur.

You must have something selected in the window for this function to work. Usually, the function is mapped with a mouse button down event.

If the optional *cut* flag is set to `1`, Arbortext Editor ignores the state of the **ctrl** key and forces a cut to take place, if it doesn't cause a context error.

To give users visual feedback, part of the selection is drawn under the mouse pointer during the drag.

-  — The green check mark indicates that you can paste the text being dragged in the current cursor location.
-  — The red not symbol indicates that you cannot paste the content being dragged in the current cursor location.
-  — The blue plus sign indicates that you can insert the content being dragged at the current cursor location.

Use the `drag_stop` function to stop the mouse tracking initiated by `drag_start`.

This function returns a `1` on success, and `0` on failure.

drag_stop

`drag_stop` (*`do_cut`*)

This function stops the drag-and-drop mouse tracking initiated by the `drag_start` function and helps the user determine whether a paste can be performed or not. The *do_cut* parameter is a user-defined variable that is set by the function. The value of the specified *do_cut* variable reflects whether the user had the **ctrl** key down when they completed the drop. If the **ctrl** key was down, the user intended to copy the selection; the specified *do_cut* will be `0`. If the **ctrl** key was up, the user intended to cut the selection, so the specified *do_cut* will be a `1`.

This function returns one of the following values:

- `0` — `drag_start` was not called.
- `-1` — The mouse pointer has not moved since `drag_start` was called.
- `-2` — The mouse pointer is not in a Arbortext Editor window.
- `-3` — Cut would cause context errors.
- `-4` — Paste would cause context errors.
- If the return code is greater than `0`, then the value is the identifier of the window that has focus. This indicates that the cursor is at a valid paste location.

drop_file_info

`drop_file_info` (*`num`*)

This function returns the path name of a file involved in a drag and drop operation. You typically call this routine from a `drop_file_over` callback the first time the callback is called for a particular drag and drop operation. If more

than one file is being dropped, then you must call this routine multiple times to obtain information about all of the files. You should only call this function once for each file during a single drag and drop operation.

The *num* argument is the number of the file for which you want to return the path name. The first file is `1`. The function returns a null string if there is no file available.

dtd_decl_path

`dtd_decl_path` (*`pubid`*`[, `*`catalogpath`*`]`)

This function returns the storage object identifier (path name) for the public identifier *pubid* specified on the first SGMLDECL or DTDDECL keyword found in a `catalog` file.

The *catalogpath* argument specifies a list of directories to search for the `catalog` file. If it is not specified, the path list specified in the *catalogpath* set option, which is initialized from the environment variable *APTCATPATH*, is searched.

The function returns a null string if neither a SGMLDECL nor a DTDDECL keyword specifying *pubid* is found in any of the catalogs searched.

dtd_tag

Note

The `dtd_tag()` function is deprecated, and support for it may be removed in a future release. `dtd_tag` is equivalent to the `oid_declared_tag()` function on page 434 which should be used in place of `dtd_tag()`

`dtd_tag`(*`tagname`*`[, `*`doc`*`]`)

This function returns `1` (True) if *tagname* is the name of an element defined in the DTD. `dtd_tag` returns `0` (False) if *tagname* is not a valid tag name or if it is a Arbortext Editor supplied tag (for example, the processing instructions `_font` or `_ignore`).

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

dupl

dupl(*s, n*)

This function returns a copy of string `s`, duplicated *n* times. For example, `dupl('*', 5)` would return `"*****"`.

edit_id

edit_id([*doc*[, *flags*]])

This function returns the current edit "id". The edit id is a number which gets incremented for each edit operation performed on the document. Command scripts can use this number to determine if the document was changed between two alias or user-defined function calls. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used. If the *flags* argument is set and has a value of `1`, then the function returns the edit id for the last structural change to *doc*.

edit_new_window

edit_new_window (*filename*)

This function opens a secondary Edit window from within Arbortext Editor. A secondary edit window allows the same functions as the Edit window.

The argument *filename* specifies the path name of a document to be loaded into the new Edit window.

Note

This function is the same as the `edit -newwindow filename` command.

elapsed_time

elapsed_time ()

This function returns the elapsed time in 100ths of a second since the current session of Arbortext Editor was started. This is the wall clock time as opposed to the CPU time (which is returned by `times`).

entity

entity(*entity*[, *doc*])

This function returns the value of the text or character entity named *name*. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used. For a text entity, the function returns the replacement value. For example, if the DTD has the definition `<!ENTITY AT "PTC Inc.">` the function entity ("AT") would return "PTC Inc..". If the entity is a character entity, this function searches the file `APSPATH/lib/charent.cf` for an entry matching *name*, returning the character given in the Unicode column if found. If it is not found, the return value varies depending on whether the document is an XML or SGML instance. For XML documents, the function returns the resolved single character replacement value, for example, for the declaration `<!ENTITY Auml "&#38;#x00C4">` the function entity ("Auml") returns "D" (character 0xc4). For SGML documents, a character entity is normally declared as a SDATA entity, for example, `<!ENTITY Auml SDATA "[Auml]">` and the function entity ("Auml") returns "[Auml]".

entity_doc

entity_doc (*entity*[, *doc*])

This function returns the document identifier of the text or file entity specified by *entity* (for the specified *doc* or the current document if *doc* is not specified). `entity_doc` returns `-1` if *entity* is not a file or text entity. If *entity* is a file entity reference, the document identifier returned will be different from the specified *doc*. If *entity* is a text entity, the document identifier returned will be *doc*.

entity_exists

entity_exists (*entity*[, *doc*])

This function returns `1` (True) if the entity name specified by *entity* is defined in the DTD or in the section of the document instance that contains document-specific markup declarations. The entity name need not start with `&'. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

entity_expand

entity_expand (*value*[, *flags*[, *doc*]])

This function expands character and text entity references in the given value string and returns the expanded string.

- *value* is the value to be expanded
- *flag* is a bitmask that controls the expansion of the entity:
  - `0x01` — expand character entity references
  - `0x02` — expand text entity references

  If omitted, the default is to expand all types of entity references.
- *doc* is the document which defines the entities. If omitted, the current document is used.

In XML the < and `&` characters must be escaped inside an attribute value; for example, in an attribute value, the string `A<B&C>D` should be encoded as `A<&C.D`.

To get the expanded value of the src attribute on the tag containing the caret:

```
local expandedValue = entity_expand( oid_attr( oid_caret(), 'src' ) )
```

entity_first

entity_first (*entity*[, *doc*])

This function returns the OID of the first element within the text or file entity specified by *entity* (for the specified *doc* or the current document if *doc* is not specified). `entity_first` the null OID if *entity* is not a file or text entity, or if there are no elements within the entity.

entity_last

entity_last (*name*[, *doc*])

This function returns the OID of the last element within the text or file entity specified by *name*, for the specified *doc* (or current document, if none given). Returns the null OID if *entity* is not a file or text entity or if there are no elements within the entity.

entity_name

entity_name (*oid*)

If *oid* is an object within a text or file entity, this function returns the name of the entity (for example, `chap1`). Returns the null string if *oid* is not contained within an entity, but is in a top-level document tree.

entity_notation

entity_notation (*name*[, *doc*])

This function returns the notation value from the declaration for the entity *name*. It returns the null string if the entity does not exist or is not a graphic entity.

The *doc* argument specifies the identifier of the document to query. If omitted or `0`, the current document is used.

entity_parfile

entity_parfile (*name*[, *doc*])

This function returns the name of the parameter file entity in which the entity was declared (if the entity was declared in a parameter file entity). Otherwise, the null string is returned.

name is the name of the entity. A leading `&` is optional for general entities. The leading percent sign `%` must be supplied for parameter entities.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

entity_path

entity_path (*name*[, *doc*])

This function returns the resolved absolute path name for the external entity given by *name*. If the entity given by *name* is not defined or is not a file or graphic entity name, the function returns the null string. The *doc* parameter specifies the identifier of the document tree to query. If omitted or 0, the current document is used. This callback should always be set globally.

For example,

```
$path = entity_path("prodname")
```

would return the full path and file name of the `prodname` file entity in the `$path` variable.

entity_pubid

entity_pubid (*name*[, *doc*])

The entity_pubid function returns the PUBLIC identifier from the declaration for the entity *name*. A leading `&` is optional for general entities. The leading percent sign `%` must be supplied for parameter entities.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

It returns the null string if the entity does not exist or if no PUBLIC identifier was specified.

entity_relative_path

`newpath=`**`entity_relative_path`**`(`*`path`*`, `*`dir`*`, `*`type`*`)`

This function converts the absolute path name given by *path* into a relative path name if possible. If the file is in or below the same directory as the supplied directory *dir*, then a path name relative to that directory is returned. If the file is not in or below *dir*, depending on the value of type, if the file is in or below a directory from one of the path list options being considered, then a path name relative to that directory is returned. Otherwise, the original input path name is returned.

The *type* parameter can have one of the following values:

- `0` if the reference is to a graphic.

  If it is `0` then the `set graphicspath` on page 828 option is considered
- `1` if the reference is to a file entity.

  If it is `1` then the `set entitypath` on page 791 option is considered
- `2` if the reference is to an XML inclusion.

  If it is `2` then both `set graphicspath` on page 828 and `set entitypath` on page 791 are ignored.
- `3` if the reference is a DITA reference.

  If it is `3` then the `set ditapath` on page 780 option is considered.

entity_resolve

`entity_resolve` `(`*`name`*`[, `*`pubid`*`[, `*`sysid`*`[, `*`catalogpath`*`]]])`

This function does a catalog lookup to resolve the entity specified by the *name*, *pubid* and *sysid* parameters, returning the system path name. If the identifier is not found in any of the `catalog` files in the search path, the `entity_resolve` function returns a null string.

The *name* argument specifies the entity name to be used in the catalog lookup. If can be a null string if the *pubid* and/or *sysid* are non-null. The name must not start with the `&` general delimiter.

The *pubid* argument specifies an optional public identifier associated with the entity declaration.

The *sysid* argument specifies an optional system identifier associated with the entity declaration.

The *catalogpath* argument specifies a list of directories to search for the `catalog` file. If it is not specified, the path list specified in the `set catalogpath` option, which is initialized from the environment variable *APTCATPATH*, is searched.

The function `public_id_path(pubid, "", catpath)` is equivalent to `entity_resolve("", pubid, "", catpath)`.

The function `entity_resolve(name, pubid, sysid, catpath)` is equivalent to `catalog_resolve("ENTITY", name, pubid, sysid, catpath)`.

entity_source

`entity_source` (*`name`*`[,` *`doc`*`]`)

The `entity_source` function returns a number indicating the origin of the declaration for the entity named *name*. The return value is one of the following integers:

- `-1` — entity not declared.
- `0` — entity defaulted.
- `1` — entity declared in declaration subset (private markup) only.
- `2` — entity declared in DTD only.
- `3` — entity declared in both DTD and declaration subset.

The argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

entity_sysid

`entity_sysid` (*`name`*`[,` *`doc`*`]`)

The `entity_sysid` function returns the SYSTEM identifier from the declaration for the entity *name*. A leading `&` is optional for general entities. The leading percent sign `%` must be supplied for parameter entities.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

It returns the null string if the entity does not exist or if no SYSTEM identifier was specified.

entity_tag

entity_tag (*tagname*[, *doc*])

This function returns 1 (True) if the markup named *tagname* is really a tag used to represent an SGML entity declaration. If the markup is not an SGML entity declaration, 0 is returned. The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

Note

If the document type definition for your document instance includes the NAMECASE GENERAL NO setting, then the *tagname* parameter will be case-sensitive.

entity_to_unicode

entity_to_unicode (*entname*)

Provided the character entity name *entname*, this function returns the unicode equivalent. For example, if you were to give *entname* the value of 'Theta', the function would return the value 920.

If there is no unicode equivalent for the character entity *entname*, the function returns a zero (0).

Note that the available character entity names depend on the document type you are using. Use the *Arbortext-path*\lib\charent.cf file a reference to character entity names.

entity_type

entity_type (*name*[, *doc*])

The entity_type function returns a string describing the type for the entity named *name*. The return value is one of the strings:

- accent — character (SDATA) entity that is an accent
- char — character (SDATA) entity
- file — external entity
- filep — parameter file entity

- `graphic` — external graphic (NDATA) entity
- `msp` — marked section parameter entity
- `text` — internal general entity

If *name* is not a valid entity, the null string is returned.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

eof

`eof``(`*`fid`*`)`

This function returns `1` if an end-of-file condition has occurred on file identifier *fid*, or `0` otherwise. *fid* must be a return value from a previous call to `open`.

error

`error``(`*`message`*`)`

This function sounds a beep and displays the error message specified by *message* in the status bar of the current window if possible, otherwise in a separate window. The message is also assigned to the *$ERROR* predefined variable.

The `error` function is used by Arbortext Editor to display most error messages.

errors_suppressed

`errors_suppressed` `()`

This function returns `1` if errors are currently being suppressed in ACL. This is usually the case when an `eval` or `execute` command is active.

eval (Function)

`eval``(`*`expr`*`[, `*`package`*`])`

This function parses and evaluates the value of the expression *expr*. The result is returned as a string or number depending on the context. When called from a user-defined function, the expression is compiled within the current scope, so local variables are accessible.

package, if specified, changes the effective package used to evaluate the expression. By default, the current package is used.

If there is a syntax or runtime error, the variable `$ERROR` gives the error message. If there is no error, then `$ERROR` is set to the null string. This function does not generate error popup messages. The caller must display the value of `$ERROR` if desired.

Here is an example of how to call a function whose name is not known until runtime (that is, the value of the variable $funcname is unknown at compilation):

```
eval("$funcname($$x)")
```

In this case, the argument is the variable *x*. The double dollar-sign is not needed if `eval` is called from the context of a user-defined function.

event_process

`event_process` (`[`*`blockall`*`]`)

This function enters a nested event loop to block the current thread of ACL execution until some until other event causes an `event_stop_process` on page 351 call. If the *blockall* parameter is specified and non-zero, then all user input events are blocked. This might be useful to block the user while waiting for input from a network channel; the callback set with `channel_set_callback` on page 976 would resume the execution thread by calling `event_stop_process`.

This function returns the value passed to the `event_stop_process` call that terminated the nested event loop.

> **Caution**
>
> This function should be used with extreme care; an application should only call `event_process` if it has a method for calling `event_stop_process`, for example, from a dialog item callback, timer callback, channel callback, or other similar callbacks.

Example:

```
function timerCallback()
{
 event_stop_process(1);
#reancel0timer
}
timer_add_callback(500, "timerCallback");
window_set(0, "message", "waiting...");
# wait 5 seconds, blocking all input
$result = event_process(1);
window_set(0, "message", "event_process returned $result");
```

event_stop_process

event_stop_process (*code*)

This function terminates the nested event loop entered by the last event_process on page 350 call. When the event that resulted in the event_stop_process call is finished, ACL execution resumes after the event_process call.

The code parameter becomes the return value from event_process.

Example:

```
function timerCallback()
{
 event_stop_process(1);
#reancel0timer
}
timer_add_callback(500, "timerCallback");
window_set(0, "message", "waiting...");
# wait 5 seconds, blocking all input
$result = event_process(1);
window_set(0, "message", "event_process returned $result");
```

execute (Function)

execute(*cmds*[, *package*])

This function parses and executes *cmds* as a sequence of Arbortext Editor commands. The result is the value of the last command executed (that is, what the $status variable is set to): 0 if success, 1 if a runtime error, or 2 if a syntax error. If there is a syntax or runtime error, the variable $ERROR gives the error message.

When called from a user-defined function, the expression is compiled within the current scope, so local variables are accessible. The argument *package* may be specified to change the effective package used to execute the command. By default, the current package is used.

This function does not generate error popup messages. The caller must display the message if desired. For example, the execute command could be written as an alias using this function:

```
alias execute {
if (execute("$*") != 0) {
 beep;
 response($ERROR)
}
}
```

exit_editor

exit_editor ([*rc*[, *code*]])

This function terminates the application with *rc*, which is a return code you assigned, or `0` if not supplied, as the exit program code. The argument *code* determines if the user is prompted for unsaved changes or not and has one of the values:

- `0` — prompt about any unsaved changes, that is, `quit`
- `1` — save all modified documents without prompting, that is, `exit`
- `2` — do not prompt about unsaved changes and quit without saving modified documents, that is, `quit ok`

The default is 0.

expand_file_name

expand_file_name (*pathname*)

This function returns the path name corresponding to *pathname*. Any environment variables in the string expression are expanded.

A leading ~ expands to the value of the *HOME* environment variable. If the *HOME* environment variable is not set, the ~ resolves to the value of the *HOMEDRIVE* and *HOMEPATH* environment variables (which are usually set on Windows). If neither *HOME* or *HOMEPATH* are set, the ~ is not substituted and will remain in the path.

Examples

```
$f = 'C:\source_files\graphics\home.tif'
g$example = expand_file_name($f)
```

file_close

file_close ([*win*[, *flags*]])

This function closes (destroys) the window specified by the *win* parameter. If *win* is not specified, it closes the current window. Executing this function is similar to choosing the **File ▸ Close** menu option.

The *flags* parameter is a bitmask specifying special file closing settings and is constructed by ORing the flags from the following list:

- `0x0001` — Set this flag to suppress all prompts to save the document open in the window.
- `0x0002` — Set this flag to close the very last Arbortext Editor window. By default, closing the very last window causes it to remain open but display an empty text document. Note that even when the very last Arbortext Editor

window is closed, the Arbortext Editor process remains running. Be careful when using this flag.

- `0x0004` — Set this flag to not offer the option to **Cancel** during any prompts to save a document.

If the document in the specified *win* (or current window) appears in both an Edit view and a Document Map view, both view "windows" will be closed. If there are other window frames open for the current session, the window frame is destroyed as well.

The first view "window" in a window frame cannot be destroyed. If the current session has only one window frame, `0x0002` was not specified, and the function closes the document in the first view "window" in the window frame, a blank ASCII document is placed in the first view window.

This function does not affect the windows in any other window frames that contain the same document appearing in the window specified by *win*.

file_directory

`file_directory` (*`path`*)

This function returns `1` (True) if the path name specified by *path* is a directory.

file_entity_names

`file_entity_names` (*`arr`*`[, `*`doc`*`]`)

The `file_entity_names` function fills the array *arr* with an alphabetical list of file entity names which are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

Entities declared in the DTD are included in the array *arr* in addition to those declared in the declaration subset.

file_entity_tag

`file_entity_tag` (*`tagname`*`[, `*`doc`*`]`)

This function returns 1 (True) if the tag named tagname is really a tag used to represent an SGML file entity declaration. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

file_is_graphic

`file_is_graphic` (*`path`*)

This function returns `1` (True) if the path name specified by *path* is a recognized graphic file, or ends with an extension specified by the `set othergraphicsextensions` option.

file_is_zip

`file_is_zip` (*`path`*)

This function returns a non-zero number (True) if the path name specified by *path* is a recognized zip archive file. It returns zero if the file is not a recognized zip archive file or on any error.

Note that this function does not detect other forms of compressed files, such as `tar` or `rar` files.

file_mtime

`file_mtime` (*`path`*)

This function returns the last modification time of the file specified by *path* or `0` if the file does not exist. The time is given in seconds since the epoch (which is operating system-specific). This time can be converted into a string for display using the `ctime` function.

The *path* argument is either a file system path or an HTTP or HTTPS resource. For the latter, the function issues an HTTP HEAD request and uses the response header called `Last-Modified` to determine the modification time. On any error, or if the desired response header is missing, this returns `0`.

Examples

```
eval ctime(file_mtime("memo.sgm"))
eval ctime(file_mtime("/Arbortext-path/memo.sgm"))
eval ctime(file_mtime('c:\memo.sgm'))
eval ctime(file_mtime("c:\\memo.sgm"))
```

Note

A backslash must be escaped if used with a double-quoted string since it introduces a C-style escape (for example, \n is a newline, \t is a tab, and so on). Alternatively, single quotes should be used since backslashes are not interpreted as escape characters.

file_newer

file_newer (*file1*, *file2*)

This function returns `1` (True) if the modification date of the file specified by *file1* is greater, or more recent, than that of the file specified by *file2*.

file_public_id

file_public_id (*path*)

The `file_public_id` function returns the public identifier for the SGML document instance *path*. It returns the null string if the file indicated by path does not exist, or does not have a public identifier.

file_selector

file_selector (*dir*[, *ext*[, *filter*[, *title*[, *flags*]]]])

This function displays a file selection dialog box and returns the user's response. The null string is returned if the user selects **Cancel**.

dir is the directory initially displayed in the dialog box. If the *dir* directory is not a valid drive, then the dialog box uses the last drive the Open dialog box used to open a file in Arbortext Editor.

The remaining arguments are optional:

- *filter* is a string specifying a list of file types to display in the **List Files of Type** pulldown list. A vertical bar "|" is used to separate each list item. There are two parts to each item, which are also separated by a vertical bar "|". The first part is the string displayed in the pulldown list. The second part is the

corresponding pattern used to filter the files displayed in the dialog box. See the example that follows.

- *ext* specifies the default extension to display in the File Name control. It should be one of the extensions given in the filter list *filter*. The extension separator "." must not be included.
- *title* specifies the window title for the dialog box.
- *flags* is a bitmask specifying special file selector settings and is constructed by ORing the flags from the following list:
  - `0x01` — Set this flag to select directories instead of files.
  - `0x02` — Set this flag to allow new files (or directories if bit `0x01` is set) to be created.
  - `0x04` — Set this flag to change the default button in the dialog box from **Open** to **Save**.
  - `0x08` — Set this flag if the *dir* parameter is a file path that should be highlighted as the chosen file when the selector is displayed.

  If no value is given, the file selector will be set to open existing files.

Example

```
l="SGML Files|*.sgm|Command Files|*.acl|All Files|*.*"
fname = file_selector(".", "sgm", l, "Open Document")
```

The `*` wildcard will include all files.

file_size

`file_size` (*`path`*)

This function returns the size in bytes of the file specified by *path*. The *path* argument is either a file system path or an HTTP or HTTPS resource. For the latter, the function issues an HTTP HEAD request and uses the response header called `Content-Length` to determine the modification time. On any error, or if the desired response header is missing, this returns `-1`.

The `file_size` function can be written using file identifier functions as follows:

```
function file_size(path)
{
 local fid = open(path, "rb")
 if (fid < 0) {
 return -1
}
 seek(fid, 0, 2); # to eof
 local off = tell(fid)
 close(fid)
 return off
```

```
}
```

file_system

file_system ([*path*])

This function returns a string which identifies the type of file system containing the file specified by *path*.The return value is "fat" if the path name specifies a DOS file system, "ntfs" if a Windows file system, and so on. If *path* is omitted, the current working directory is used.

filename_encode

filename_encode (*name*[, *escape*[, *translateSlash*]])

This function returns the string *name* encoded according to internet RFC 1738. Most non-printable characters in *name*, including all characters with ASCII values over 255, are encoded as 2 or 4 digit hexadecimal sequences following an escape character.

This function has the following parameters:

- *name* — The character string to encode.
- *escape* — Optional. Specifies the character to note the start of the hexadecimal representation of an encoded character. If *escape* is omitted or specified as the empty string, `filename_encode` uses the `%` (percent) character. If *escape* contains more than one character, only the first character is used.
- *translateSlash* — Optional. If *translateSlash* has any value other than `0` or "" (the null string), backslash (Windows) separators (`\`) are converted to forward slashes (`/`) and forward slashes are not encoded. If the parameter is omitted, or has a value of either `0` or the null string, backslashes are not converted to forward slashes, and both backslash and slash characters are encoded.

  For example:

  `filename_encode('a\b/c','_',0)` returns a_5Cb_2Fc

  `filename_encode('a\b/c','_',1)` returns a/b/c

  translateSlash can only be specified when *escape* is also specified.

The calls:

```
filename_encode("abc def");
filename_encode("abc def", "");
filename_encode("abc def", "%");
filename_encode("abc def", "%XYZ");
```

Return:

```
abc%20def
```

The call:
```
filename_encode("abc def", "_")
```
Returns:
```
abc_20def
```
The call:
```
filename_encode( "abc def\", "%", 1)
```
Returns:

```
abc%20def/
```

filename_to_url

filename_to_url (*path*)

This function returns the file name *path* converted to a UTF8–encoded file-scheme URL conforming to RFC 1738. It converts the file name to an absolute path name, replaces all file separator characters with '/', and encodes all non-alphanumeric characters (except for `$`, `-`, `_`, `.`, `+`, `!`, `*`, `'`, `(`, `)`, `/`, and `:`) into a sequence of one or more 3-character strings `%`*XX*, where the *XX* values are the two digit hexadecimal codes for the UTF-8 encoding of the character. It returns the result prefixed with `file://`. If the file name is a directory, a trailing `/` is appended to the result.

Note

The file name should not include a fragment or anchor identifier (for example, Xpointer) specification; but this can be appended to the result.

For example, the following function:
```
filename_to_url('article#1.xml')
```
would return a string like the following:
```
file:///C:/documents/article%231.xml
```
As another example, the function:
```
filename_to_url('c:\file' . chr(0x3042) . '.txt')
```
would produce:
```
file:///C:/file%E3%81%82.txt
```

filep_entity_names

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

The `filep_entity_names` function fills the array *arr* with an alphabetical list of file parameter entity names which are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

Only entities declared in the declaration subset are included in the array *arr*.

find

find(*searchStr*[, *flags*[, *element*[, *doc*]]])

This function searches for the specified string *searchStr*. The search ignores generated text and returns 1 on success, 0 on failure.

- *searchStr* — String to search for.
- *flags* — Bit mask for modifying the search.
  - `0x0001` - backward search (*-b*)
  - `0x0002` - case sensitive search (*-c*)
  - `0x0004` - *searchStr* is regular expression (*-e*)
  - `0x0008` - whole word only search
  - `0x0010` - *searchStr* contains markup (*-markup*)
  - `0x0020` - quiet search (*-q*) default is (*-noq*)
  - `0x0040` - don't prompt for wrapping at end of instance (*-wrapprompt*)
  - `0x0080` - wrap search (*-wrapscan*)
  - `0x0100` - don't wrap search (*-nowrapscan*) default is `-wrapscan` preference
  - `0x0200` - select *searchStr* when found (*-select*)
  - `0x0400` - don't select *searchStr* when found (*-noselect*) default is `-selectscan` preference
  - `0x0800` - search contents of entities (*-entityscan*)
  - `0x1000` - don't search contents of entities (*-noentityscan*) default is `-entityscan` preference
  - `0x2000` - search for next equation (*-equation*) *searchStr* ignored

- `0x4000` - search for next image (*-image*) *searchStr* ignored
- `0x8000` - search for next table (*-table*) *searchStr* ignored
- `0x10000` - selection will be unbalanced regardless of the `balancedselections` on page 758 Advanced Preference and `set` option.

- *element* — Search only the contents of elements with this name.
- *doc* — Current document instance is used if not specified.

find_dita_rd_dcf

`find_dita_rd_dcf` (*`rdType`*`[, `*`doc`*`]`)

This function returns the document type definition (`.dcf`) file that will be used by a resolved document generated for the given DITA map. The *rdType* parameter is the type of resolved document for which you want to determine the `.dcf` file. Allowed values are `ditaRDStyle` for the resolved document for styling and `ditaRDEdit` for the resolved document for editing. The *doc* parameter is the DITA map that would be used to generate the resolved document.

find_file_in_path

`find_file_in_path` (*`filename`*`[, `*`mode`*`[, `*`pathlist`*`[, `*`defaultpath`*`]]])`

This function searches the list of directories specified by *pathlist* for the file given by *filename* returning the path name of the first match if found.

The *mode* parameter specifies the permissions for the file and is one of the following characters:

- `e` — exists
- `r` — read
- `w` — write
- `x` — executable

If *mode* is omitted, `e` is assumed.

If *pathlist* is omitted, the value of the `PATH` environment variable is used. Each directory component in *pathlist* is expanded using `expand_file_name` so home directory and environment variable references are permitted.

The *defaultpath* parameter specifies the default value for a `%D` qualifier given as part of *pathlist*, that is, the value of *defaultpath* will be substituted for `%D`. If omitted, no substitution is done. See the third example below.

The function returns the null string if *filename* is not found within the pathlist, or is not accessible according to *mode*.

Examples

```
browser = find_file_in_path("Netscape");
logofile = find_file_in_path("logo.gif", "r", option("graphicspath"));
cfile = find_file_in_path("catalog", "r", ".${main::PCS}%D", \
 option("catalogpath"));
```

flush

flush(*fid*)

This function flushes any output buffered for the file identifier *fid*, which must be a return value from a previous call to `open`. `flush` returns `0` on success or `-1` on failure (for example, if the associated file was not opened for writing).

flush_dita_rdgen_cache

flush_dita_rdgen_cache()

This function deletes all temporary DITA resolved document for styling DTDs and resolved document for editing XML schemas and unloads the internal data structures representing those document types. All future requests to retrieve a resolved document will result in a new file being generated.

font_families

font_families (*arr*)

This function fills the array *arr* with an alphabetical list of font family names available for display on the current system. The first name returned is stored at `index 1`. The function returns the number of font family names added to the array.

All previous elements in *arr* are discarded when the function executes.

formatting

formatting ()

This function returns `1` (true) if a `format`, `print`, or `preview` command is currently running. Otherwise, `formatting` returns `0`.

forward_char

forward_char (*count*[, *doc*])

This function moves the cursor *count* characters to the right in the document given by *doc*, or the current document if *doc* is omitted. If *count* is negative, then the logical cursor is moved left *count* characters. Each non-text object (that is, tags, equations, graphics) counts as one character. This function returns `1` (True) if the cursor was moved the specified distance. It returns `0` (False) if the end (or beginning) of the document was reached. Note, that `forward_char(1)` is similar to the command `caret 0,+1`, except that the function always moves one character.

The `caret` command treats the case of the cursor at the end of a screen line specially and moves the cursor to the beginning of the next line, which is before the same character. Because of this, for cases where the screen representation is not important or relevant (for example, for a document loaded with `doc_open`), `forward_char` is more useful.

fosi_public_id

fosi_public_id (*fdoc*)

This function returns the public identifier for the instance document associated with the FOSI document given by *fdoc*, or the null string if *fdoc* is not an attached FOSI.

fosivar_value

fosivar_value (*window*, *varname*, *oid*)

You can call `fosivar_value` from functions that are called as a result of the `attloc=system-func` FOSI setting.

The `fosivar_value` function works the same as #FOSI, except that it can be called from within functions invoked by the `system-func` feature.

window is the window ID that was passed as an argument to the function called by the `system-func` function. The value of this parameter can be `-1`.

varname is the name of the FOSI string variable or counter variable whose value is to be returned.

oid is the object identifier (OID) of the element in the user's document whose e-i-c is being processed

framesets

framesets (*descriptionArr*, *locationArr*[, *doc*])

The `framesets` function fills the *descriptionArr* array with frameset descriptions and the *locationArr* array with the paths and file names of framesets declared in the document type configuration file (`.dcf`) for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

The arrays are returned in the order in which the framesets are declared in the `.dcf` file. The function also returns the number of entries in each array. The first entry is stored at index 1.

function_argc

`function_argc` (*`name`*`[, `*`varname`*`]`)

The `function_argc` function returns the number of arguments with which function *name* is declared. `-1` is returned if *name* is not a known function.

If *varname* is given, the minimum number of arguments is assigned to that variable. *varname* must be either a scalar variable or array element.

`function_argc` works for both built-in functions and those defined in ACL. If name is an ACL function not yet loaded but declared via autoload, `function_argc` will cause the function to be loaded to return the actual argument counts.

function_file

`function_file` (*`name`*`[, `*`load`*`]`)

The `function_file` function returns the path name of the file which defines the function *name*. If *name* is a built-in function, the string `built-in` is returned. If *name* is not a known function, `function_file` returns the null string.

If *name* is an ACL function not yet loaded but declared via autoload, `function_file` will return the null string by default. If the optional parameter *load* is specified and has a non-zero value, `function_file` will cause the function to be loaded to return the actual path name.

function_names

`function_names` (*`arr`*`[, `*`package`*`]`)

The `function_names` function fills the array *arr* with a list of all functions defined in the specified *package* and returns the number of functions.

If *package* is omitted, the list of all built-in functions is returned.

generate_id

generate_id (*doc*, *oid*, *attrname*[, *text*[, *flags*]])

The generate_id function generates an ID and optionally adds a suffix to the ID to make it more likely that the ID will be unique. The function will generate the ID through one of the following methods:

- Call the generate_id on page 993 callback, if it is available.
- Generate an initial ID based on title, text, and element information from the document. If a unique ID is called for, append a string to the initial ID value. The string will be a hyphen (-) followed by eight hexadecimal characters representing the current time and date.

The generate_id function has the following arguments.

- *doc* — (Optional.) The identifier of the document for which the ID is being generated. A value of 0 (zero, the default) refers to the current document.
- *oid* — (Optional.) The object identifier of the element for which the ID is being generated. When an ID is being generated in a context that does not involve a particular element, the value of *oid* will be oid_null().
- *attrname* — (Optional.) The name of the attribute for which the ID is being generated or a null string when the ID is not being generated for use as an attribute value.
- *text* — (Optional.) An initial text string that can be used to help generate the ID. For example, this text could be used when the function is called without valid *doc* or *oid* arguments.
- *flags* — (Optional.) A bitmask that determines whether the suffix is added to the ID.
  - 0x000 — (The default value.) The generateuniqueid set option and Advanced Preference is used to determine if a unique suffix is added.
  - 0x001 — Always add a unique ID suffix.
  - 0x002 — Never add a unique ID suffix.

  Other bits are unused and should not be set. If bits 0x001 and 0x002 are set, a value of 0x000 is assumed.

get_appdata_dir

get_appdata_dir ([*subpath*[, *flags*]])

When called with no arguments, this function returns the full path to the application data directory.

This directory typically is: `C:\Documents and Settings\<login>\Application Data\PTC\Arbortext\Editor`

The *APTAPPDATADIR* environment variable can be used to override the default location.

- *subpath* — Optional. A relative directory structure such as `onedir\twodir`. When *subpath* is given, the function returns a full path to the given *subpath* of the application data directory.
- *flags* — Optional directory options. The following bit values are supported in the *flags* parameter.
  - `0x0001` — Ensure the returned directory structure is fully created
  - `0x0002` — Causes `get_appdata_dir` to return a local (non-roaming) application data directory such as: `C:\Documents and Settings\<login>\Local Settings\Application Data\PTC\Arbortext\Editor`

If errors are encountered, the return value will be an empty string and *main::ERROR* will be contain an error message.

Example:

```
path = get_appdata_dir('acme/customizations', 0x01);
```

get_cache_dir

`get_cache_dir` `()`

Returns the location of the Arbortext Editor cache directory (`.aptcache`).

get_composer_log_contents

`get_composer_log_contents``([`*`html`*`])`

The `get_composer_log_contents` function returns the contents of the log file associated with the last publish operation as a formatted multiline string. If the *html* parameter is present and non-zero, the string is HTML wrapped in a `pre` element.

> **Note**
>
> This function doesn't return anything if the Event Log window is closed after a publish operation.

get_composer_log_doc

get_composer_log_doc ()

The get_composer_log_doc function returns the ID of the read-only document containing the log output from the last publish operation.

get_custom_dir

get_custom_dir ([*app_or_index*])

The get_custom_dir function returns the full path of the directory in which a given application is installed. An application is not required to be installed inside the Arbortext Editor or Arbortext Publishing Engine install tree, as long as you use the *APTAPPLICATION* or *APTCUSTOM* environment variable to specify another path or path list to search for custom applications (using a file system specification for *APTAPPLICATION*; HTTP references are not supported).

In cases where *APTCUSTOM* cites a zipped custom directory, application, or CMS adapter, get_custom_directory returns a full path to the locally expanded form of the zipped customization. The expanded form is stored in the Arbortext Editor cache (.aptcache\zc) directory. For example, if *APTCUSTOM* is set to the following value:

```
http://server/apps/custom.zip
```

The get_custom_directory function returns a path similar to the following value:

```
C:\Documents and Settings\user\Local Settings\
 Application Data\PTC\Arbortext\Editor\.aptcache\zc\1
```

If *APTCUSTOM* points to a zipped application called com.acme.app and is set to the following value:

```
http://server/apps/app.zip
```

The get_custom_directory(com.acme.app) function returns a path similar to the following value:

```
C:\Documents and Settings\user\Local Settings\
 Application Data\PTC\Arbortext\Editor\.aptcache\zc\3\com.acme.app
```

The *app_or_index* parameter is optional; if omitted, 0 is assumed. If specified, it can be one of the following:

- The application's uniquely named directory (usually a Java class name, such as com.arbortext.sample).

  The function returns the full path of the directory containing the given application. An empty string is returned if no matching application is found.

  Example:

  ```
  local dir = get_custom_dir("com.arbortext.sample");
  ```

- An integer greater than or equal to zero.

The function returns the full path of the `custom` or `application` directory corresponding to the integer specified. The function follows the search order of the `custom` and `application` directory path list determined when Arbortext Editor or the Arbortext Publishing Engine starts. For example, the order would be `0` for the first directory in the search order list, `1` for the next directory, and so on. This function returns an empty string for an index that is out of range.

The search path is built at startup from any `custom` directories as well as any applications loaded from the `application` directory.

Example:

```
local i;
local dir;
for (i = 0; dir = get_custom_dir(i); i++) {
 # Do something with the directory here.
}
```

- A negative integer.

  The function returns the full path of the `custom` or `application` directory corresponding to the integer specified. The function reverses the search order of the `custom` and `application` directory path list determined when Arbortext Editor or the Arbortext Publishing Engine starts. For example, the order would be `-1` for the last directory in the search order list, `-2` for the next to last directory, and so on. This function returns an empty string for an index that is out of range.

  The search path is built at startup from any `custom` directories as well as any applications loaded from the `application` directory.

  Example:

```
local i;
local dir;
for (i = -1; dir = get_custom_dir(i); i--) {
 # Do something with the directory here.
}
```

get_custom_property

`get_custom_property` (*`key`*`[,` *`default`*`]`)

The `get_custom_property` function retrieves the value of a global parameter or variable that has been configured in an `application.xml` file.

The function returns the value for the name of the variable or other property specified by *key*. The return value is the Unicode string associated with the specified *key* (as defined in the `application.xml` file). This value may be an empty string if no value is configured. Specify *key* using the following format:

application.qualified.name.property-name

For example, if the application's unique full name is `com.arbortext.sample` and the preference name specified in `application.xml` is `Color`, then you would specify `com.arbortext.sample.Color`.

The optional *default* parameter specifies a value to substitute if the *key* parameter name is not specified in the `application.xml` file. If there is no matching parameter for the specified *key* as defined by `application.xml`, the *default* parameter value (if specified) will be returned instead.

get_default_printer

`get_default_printer` `()`

This function returns the name of the default printer, or the empty string if no default printer is available.

get_newlist_entry

`get_newlist_entry` (*`entry`*, *`array[]`*)

Given a new dialog entry (name, description, or path), this function returns an associative array containing information about the entry. The function return code is 1 for success, 0 for failure.

- *entry* — String indication entry. It is the base name of the document type DTD (e.g. article), the full DTD path (for example, *APTPATH*`/doctypes/article/axdocbook.dtd`), or the description that appears in the **New Document** dialog box (for example, **Arbortext XML DocBook V4.0**).
- *array* — Associative array that will contain entry information on success. The indexes are:
  - `name` — Name of document type.
  - `description` — Description of document type as appears in the **New Document** dialog box.
  - `dtdpath` — The path to the document type DTD.
  - `template` — The path to the document type template as specified in the DCF file.
  - `sample` — The path to the document type sample as specified in the DCF file.
  - `descriptionindex` — Ordinal indicating the position of the description in the DCF file.
  - `displayed` — Boolean that indicates if this document type is currently being displayed in the **New Document** dialog box.

- `category` — The category in which the document type appears in the **New Document** dialog box.

get_num_printers

`get_num_printers` `()`

This function returns the number of available printers. This number is determined by querying the operating system for the set of installed printers.

get_preferences_path

`get_preferences_path` `([write])`

This function returns the path name of the user's preferences file. If the optional *write* parameter is present and not zero, the function returns the path name that will be used to write the preferences file. This may be different than the path name used to read the preferences file if an upgrade has been performed. For example, an old `epic.wcf` may still be read and a new `arbortext.wcf` may be written to.

get_printers

`get_printers` `(*(printers[])*)`

This function populates the array 'printers' with the names of the available printers.This list is determined by querying the operating system for the set of installed printers.

get_user_property

`get_user_property` `(*key*[, *default*])`

The `get_user_property` function returns the value for a property that has been set in one of the following ways:

- a value specified by the `set_user_property` function on page 515.
- a value subsequently stored in the `arbortext.wcf` preferences file after being set initially by the `set_user_property` function.

The return value is the Unicode string associated with the user preference or other property specified by *key*. This value may be an empty string if no value is configured.

Specify *key* using the following format:
`application.qualified.name.property-name`

For example, if an application's fully qualified name is `com.arbortext.sample` and the user setting is `windowVisible`, then you would specify the *key* parameter as `com.arbortext.sample.windowVisible`.

The optional *default* parameter specifies the value to substitute if the *key* parameter name is not specified. If there is no matching parameter, the *default* parameter value (if any) will be returned if specified.

getline

`getline``(`*`fid`*`[, `*`varname`*`])`

This function reads the next line from the file associated with the file identifier *fid*, which must be a return value from a previous call to `open`.

fid must refer to a file that was opened for reading.

varname must be either a scalar variable or an array element. If varname is given, then the line is placed in the variable by that name. The return value in this case is the number of characters read. If the end of the file is reached before reading any characters, `getline` returns `0` and sets *varname* to the null string. If *varname* is not given, then `getline` returns the line or a null string if the end of file has been reached.

- `getline` does not remove the terminating newline character. If desired, use the function `chop` to do this.
- `getline` has an internal limit of 3000 characters. If a newline character is not found within the limit, then that many characters are returned without the terminating newline character.
- `getline` does not support reading lines that contain null characters (\000), that is, binary data, because it is implemented using the C language function `fgets`. The `read` function can be used to read binary data.

Examples

The following is an example of a function to copy one file to another, using `getline` and `put`:

```
function fid_copy(from, to)
 {
 local inf, outf, line
 inf = open(from, "r")
 if (inf < 0) {
 eval "Couldn't open file for read:", from
 return 0
}
 outf = open(to, "w")
 if (outf < 0) {
 close(inf)
 eval "Couldn't open file for write:", to
```

```
 return 0
}
 while (getline(inf, line)) {
 put(outf, line)
}
 close(outf)
 close(inf)
 return 1; # True
}
```

getlocale

getlocale ([*full*])

This function returns a string of the current system locale name. You will need the system locale if you are creating localized message files.

If the optional *full* parameter is present and non-zero, the function returns the full locale name (for example, English_United States.1252) as opposed to the shorter form of the locale name (for example, ENU).

Execute the command below at the Arbortext Editor command line to display the system locale name.

```
eval getlocale()
```

getpid

getpid()

This function returns Arbortext Editor process ID which can be used for constructing unique file names.

glob

glob(*filepat*[, *arr*])

This function fills the array *arr* with a list of file names that match the pattern specified by *filepat*. It returns the number of files matching the pattern or zero if nothing matched. The file names are returned in alphabetical order.

Currently, if the file pattern contains any directory components, only the last component may contain the file match patterns *, ? and [].

Examples

```
glob("*.sgm", doclist)
```

goto_cell

goto_cell (*row*, *col*)

This function moves the cursor to the end of the cell given by *row*, *col* where the upper left cell is at row 1, column 1. Returns 1 if successful, 0 if there is no such cell in the current table, or -1 if the cursor is not in a table.

goto_oid

goto_oid (*oid*[, *pos*])

This OID function moves the cursor to the position given by *oid*, *pos*. goto_oid does not reposition the screen when moving the cursor.

pos can be any of the following values:

- -1 — The cursor is placed immediately before the end tag of the element identified by *oid*.
- -2 — The cursor is placed immediately before the start tag.
- -3 — If *oid* is a start tag the cursor is placed immediately after the end tag. Otherwise the cursor is placed at the end of the object.

If *pos* is omitted, the cursor is placed after the start tag identified by *oid*.

graphic_attr_name

graphic_attr_name (*tagname*, *role*[, *doc*])

This function returns the name of the graphic *tagname* attribute for the graphic trait *role* defined in the document type's .dcf file.

Note

In SGML, if the SGML declaration for the document type definition includes the NAMECASE GENERAL NO setting, then the *tagname* parameter is case-sensitive. In XML, the *tagname* parameter is always case-sensitive.

The optional *doc* argument specifies the identifier of the document to query. If omitted or 0, the current document is used.

Valid values for *role* correspond to graphic element attributes that are set in the document type's .dcf file. The following table associates the valid values for *role* to the graphic attributes in the .dcf file.

Valid *role* values associated with graphic attributes

Value for *role*	Attribute in `.dcf` file
croph	<Graphic cropHeight="AttributeName">
cropllx	<Graphic cropLowerLeftX="AttributeName">
croplly	<Graphic cropLowerLeftY="AttributeName">
cropw	<Graphic cropWidth="AttributeName">
entity	<Graphic entity="AttributeName">
filename	<Graphic filename="AttributeName">
horzadjamt	<Graphic horizOffsetAmount="AttributeName">
horzadjpct	<Graphic horizOffsetPercent="AttributeName">
horzscale	<Graphic horizScale="AttributeName">
notation	<Graphic notation="AttributeName">
proctype	<Graphic processor="AttributeName">
reprodep	<Graphic reproDepth="AttributeName">
reprowid	<Graphic reproWidth="AttributeName">
res	<Graphic resolution="AttributeName">
scalefit	<Graphic scaleToFit="AttributeName">
vertadjamt	<Graphic vertOffsetAmount="AttributeName">
vertadjpct	<Graphic vertOffsetPercent="AttributeName">
vertscale	<Graphic vertScale="AttributeName">
view	<Graphic view="AttributeName">

If *tagname* is not defined as a graphic element in the `.dcf` file, or if no attribute is assigned to the specified *role*, this function returns the null string.

Note

Executing the function `graphic_attr_name(`*tagname*`, "entity")` returns the same value as the function `graphic_entity_attr_name(`*tagname*`)`. Similarly, the function `graphic_attr_name(`*tagname*`, "filename")` returns the same value as the function `graphic_file_attr_name(`*tagname*`)`.

graphic_entity_attr_name

graphic_entity_attr_name(*tagname*[, *doc*])

This function returns the name of the graphic entity attribute for the graphic element specified by *tagname* that is defined in the document type's `.dcf` file. If an entity attribute is not defined for the *tagname* element, `graphic_entity_attr_name` returns the null string. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter is case-sensitive.

Note

Executing the function `graphic_attr_name(`*tagname*`, "entity")` returns the same value as the function `graphic_entity_attr_name(`*tagname*`)`.

graphic_entity_names

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

The `graphic_entity_names` function fills the array *arr* with an alphabetical list of graphic entity names that are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

Entities declared in the DTD are included in the array *arr* in addition to those declared in the declaration subset.

graphic_entity_tag

graphic_entity_tag (*tagname*[, *doc*])

This function returns `1` (true) if the tag specified by*tagname* represents an SGML graphic entity declaration. The *doc* argument specifies the identifier of the document tree to query. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

graphic_file_attr_name

`graphic_file_attr_name``(`*`tagname`*`[, `*`doc`*`])`

This function returns the name of the graphic file name attribute for the graphic tag specified by *tagname*. If there is no file name attribute defined for *tagname*, `graphic_file_attr_name` returns the null string. The *doc* argument specifies the identifier of the document tree to query. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

The function `graphic_attr_name(`*`tagname`*`, "filename")` function returns the same value as the `graphic_file_attr_name(`*`tagname`*`)` function.

graphic_format

`graphic_format` `(`*`path`*`)`

This function returns the graphic file format of the graphic identified by the *path* parameter. The value *path* must be the absolute path to a graphics file. If the *path* does not identify a graphic, the function returns an empty string.

The returned value can be one of the following formats:

- `sunbm` — Sun Bitmap
- `xbm` — X Bitmap

- `eps` — Encapsulated PostScript file
- `cur`— Windows Cursor file
- `drw` — IslandDraw DRW file
- `sunicon` — Sun Icon file
- `tif` — Tag Image File Format
- `drwunc` — IslandDraw DRAW file (not compressed)
- `png` — Portable Network Graphics
- `wmf` — Windows metafile
- `icon` — Windows icon
- `pcx` — PC Paintbrush File Format
- `bmp` — Windows Bitmap
- `cgm` — Computer Graphics Metafile
- `gif` — Graphic Interchange Format
- `jpg` — Joint Photographic Exports Group
- `calsG4` — CALS raster (G4)
- `svg` — Scalable Vector Graphics
- `idr` — Arbortext IsoDraw file
- `idrz` — Arbortext IsoDraw file
- `iso` — Arbortext IsoDraw file
- `isoz` — Arbortext IsoDraw file
- `edz` — Creo View file
- `pvz` — Creo View file

graphic_information

`graphic_information` (*`image`*, *`arr`*)

This function fills the array *arr* with a list of attributes of the specified graphic, *image*, where *image* is the full path and file name of the graphic. `graphic_information` returns `1` if *image* exists, otherwise, it returns `0`.

arr is filled with the following values:

- `arr['format']` contains the name of the graphic format. If Arbortext Editor does not recognize the format, `arr['format']` will be `unknown`.
- `arr['resolution']` contains the resolution value (in dpi) if the file has a specified resolution. Otherwise, `arr['resolution']` contains `0`.

- `arr['width']` contains the width of the image.
- `arr['height']` contains the height of the image.

graphic_relative_path

graphic_relative_path(*path*[, *doc*])

This function converts the absolute path name given by *path* into a relative path name if possible. If the graphic file is in or below the same directory as the document *doc*, then a path name relative to the document directory is returned. If the file name is located within a directory given by the `graphicspath` option, or a subdirectory of a directory given by the `graphicspath` option, then the base name is returned. Otherwise, the input path name is returned.

If *doc* is zero or omitted, the current document is used.

This function is used by `oid_set_graphic_pathname` and `insert_graphic_file` functions when storing a graphic file reference.

graphic_tag

graphic_tag (*tagname*[, *doc*])

This function returns `1` if the tag *tagname* is declared as a graphic element in the `.dcf` file for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

> **Note**
>
> Arbortext Styler defines a graphic in a stylesheet by assigning the Graphic style to an element that is configured as a graphic in the `.dcf` file. Configuration changes made to the element in Arbortext Styler, for example changing its style to something other than Graphic, override the settings in the .dcf file. This may affect the working of this function.

graphic_tag_name

graphic_tag_name ([*doc*[, *flags*]])

This function returns the name of the primary graphic element defined in the document type's `.dcf` file or .style file for the document specified by *doc*. The function returns the null string if no graphic tag was designated for the document type or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

The following flags can be set:

Value	Description
`0x01`	Prompt for the graphic tag name if there are more than one in context (only applies if bit `0x04` is also set).
`0x02`	Only include graphic tags that use an entity attribute (only applies if bit `0x04` is also set).
`0x04`	Only return graphic tags that are valid in the current document context.

graphic_viewer

graphic_viewer (*path*)

This function returns the name of the embedded ActiveX dialog used to display the intelligent graphic identified by *path*. The value of *path* must be an absolute directory path.

If the intelligent graphic is an EDZ graphic and displayed by ProductView, the function returns `pview`. If the intelligent graphic is an ISO graphic and displayed by Arbortext IsoView, the function returns `iview`. If *path* does not reference an intelligent graphic, the function returns an empty string.

graphic_views

graphic_views (*path*, *arr*)

This function fills the array *arr* with all of the views that are defined in the intelligent graphic file identified by *path*. A view is a custom display of the intelligent graphic that was created and saved inside of the graphic. If *path* exists and is an intelligent graphic file, the function returns the size of the array. If *path* does not exist or is not an intelligent graphic file, the function returns `0`.

The value of *path* must be an absolute directory path.

gsub

gsub(*regexp*, *repl*[, *string*,*lvalue*])

This function globally replaces all substrings of *string* that match the regular expression *regexpr* and replaces the substring with the string *repl*. The resulting string is assigned to *lvalue*, which must be a variable name or array element. If

string is omitted, then the value of *lvalue* is used as the source string. When copying the *string* to *lvalue*, the part that was matched by *regexpr* is replaced according to the string *repl*.

If *repl* contains a "&" or "\0", then it is replaced with the substring that matched *regexpr*. If *repl* contains a "*n*", where *n* is a digit between 1 and 9, then it is replaced with the substring that matched the *n*th parenthesized subexpression of *regexpr*. *n* must not be greater than the number of parenthesized expressions in *regexpr*.

`gsub` returns the number of substrings that were replaced or `0` if no match occurred. The *lvalue* is not changed if `0` is returned.

Note

Because both the regular expression parser and the ACL parser interpret backslash sequences in strings, you must use at least two backslashes to match or substitute a single "\" character in regexp or repl. If you use double quotes to delimit string constants, you must use quadruple backslashes to match or substitute a single "\".

Examples

For the example, where `$x = "this":`

```
gsub('t', 'T', $x), $x results in "This".
```

In another example, after

```
gsub('\((a)(b)\)', '(\2\1)', "(ab)c(ab)c", $r)
is executed, the value of $r is (ba)c(ba)c.
```

hex

hex (*expr*)

This function returns the decimal value of *expr* interpreted as a hex string. For example, the value of `hex("f")` is `15`.

hidden_tag

hidden_tag (*tagname*[, *doc*])

This function returns `1` if the tag specified by *tagname* is declared as a hidden element in the document type configuration file (`.dcf`) for the document type associated with *doc*. If *doc* is `0` or omitted, the current document is used.

A hidden tag is one that does not display in Arbortext Editor dialog boxes that provide lists of DTD tags , such as the **Find Tag/Attribute** dialog box). However, a hidden tag is returned when you request a list of elements in a document type are requested (for example, when calling the `tag_names` function on page 541). Also, hidden tags are automatically excluded by the `exclude_tag` callback on page 993.

high_bound

`high_bound` (*`arr`*)

This function returns the maximum subscript allowed for the array *arr*. If the array was not declared a fixed size with the `local` or `global` statement, then `high_bound` returns the highest subscript used. If *arr* is an associative array, the function returns a null string.

hook_call

`hook_call` (*`name`*`[, `*`arg1`*`[, `*`arg2`*` ...]])`

This function calls the ACL hook function specified by *name*. The remaining arguments, if any, are passed to the hook function.

The function returns `0` if all functions on the hook function list succeed, `-1` if a function in the hook function list aborts the operation, and `-2` if there is an error.

Example

```
hook_call("preferencehook", win)
```

hook_eval

`hook_eval` (*`hookname`*`[, `*`arg1`*`[, `*`arg2`*` ...]])`

This function calls the ACL hook function specified by *hookname*. The remaining arguments, if any, are passed to the hook function.

The function returns the value returned by the called hook function. If more than one function is registered for the hook, `hook_eval` returns the return value of the last function called.

Example:

```
newpath = hook_eval("graphicpathhook", graphicfile, oid)
```

htmldoc

`htmldoc``(`*`doc`*`[, `*`source_loc`*`[, `*`css_mode`*`[, `*`encodingstring`*`]]])`

This function creates, in memory, a virtual HTML document conforming to the html DTD. The generated document is based on the current FOSI. It uses the print FOSI, if set. If a print FOSI is not set, it uses the Editor FOSI.

Upon successful completion, it returns the document ID for the virtual HTML document. If an error occurs, it returns a `-1`.

The document identifier *doc* specifies the source document.

Set *source_loc* to `1` (true) to add the `srctreeloc` attribute to the elements in the virtual HTML document. The `srctreeloc` attribute value is a tree location string indicating the element's location in the source document. The default is **0**.

Specify *css_mode* to set the level of Cascading Stylesheet (CSS) information you want to incorporate in the HTML document. Following are the valid parameters and what they generate:

- *0* — No CSS mode. Generates HTML without any CSS style or class information. This is the default.
- *1* — CSS style. Generates HTML with CSS properties specified within the `style` attribute of various HTML elements, but no classes are defined or used.
- *2* — CSS classes. Generates HTML with CSS properties specified by classes only. These classes are defined within the content of the HTML `<style>` element, and are referred to within the `class` attribute of various HTML elements.

> **Note**
>
> This parameter may not convey as much style information as with CSS modes *1* and *3*.

- *3* — CSS style and classes. Generates HTML with CSS properties specified by classes that are defined within the HTML `<style>` element, and augmented by CSS properties specified within the `style`attribute of various HTML elements.

Set *encoding* to a string that specifies the encoding for the HTML document. The following encodings are supported:

- ISO-8859-1
- ISO-8859-2
- ISO-8859-5
- ISO-8859-7
- ISO-8859-9

- Windows-1252
- UTF-8

Note

The encoding you specify should match the *encoding* parameter specified for the `write` command on page 728.

To display the HTML document, create a new window and perform a *doc_show* using the document ID that was returned.

Note

If the HTML document incorporates CSS information, you must use a browser that supports HTML version 3.2 and above to view it.

http_cache_flush

`http_cache_flush` `([url])`

`http_cache_flush` clears one or more files from the URL cache. If the optional *url* parameter is omitted, the function clears the entire URL cache except for any WebDAV URLs that are currently locked. If the optional *url* parameter is provided, the function removes the file corresponding to the specified URL (unless it is a WebDAV URL that is currently locked). The *url* parameter must be included as the remote URL reference and not the actual file name within the local URL cache.

`http_cache_flush` returns one (1) on success. If the optional *url* parameter is provided but does not correspond to any file in the URL cache, `http_cache_flush` returns a zero (0).

The following example removes all files from the URL cache and stores the return value in the `$ret` variable:

```
$ret = http_cache_flush()
```

The following example removes the cached file for the URL `http://www.company.com/files/plan.ent` and stores the return value in the `$ret` variable:

```
$ret = http_cache_flush("http://www.company.com/files/plan.ent")
```

in_context

in_context (*tagname*[, *doc*[, *flags*]])

This function determines if the tag specified by *tagname* can be inserted at the cursor. If markup is selected, this function determines if the tag can be inserted surrounding the selected markup.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

- The *tagname* string may be delimited by the start and end tag delimiters (< and >) but need not be in most cases. Special cases include the following:
  - `"text"` — Tests whether text may be inserted at the cursor.
  - `"tab"` — Tests whether a tab is valid (the latter only applies to certain document types).

  If the document type has a tag named `text` or `tag`, use `"<text>"` or `"<tag>"` to test it.
- The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

- The *flags* parameter is an optional bitmask `0x001`, which takes auto-insertions into account (those specified by `InsertAroundToFix` in the `.dcf` file). If *flags* is not supplied, `0` is used.

`in_context` returns one of the following integers:

- `0` — The tag cannot be inserted.
- `1` — The tag can be inserted.
- `2` — The tag can be inserted after modifications. For example, after adding required tags.

in_context_list

in_context_list (*arr*[, *ct*[, *oid*[, *pos*[, *filter*[, *alias*]]]]])

This function fills the array *arr* with a list of element names that are valid for `insert_tag` (or `change_tag` if *ct* is `1`) at the point given by *oid*, *pos*. If *oid* is omitted, the cursor position in the current document is used. `in_context_list` returns the number of elements stored in an array.

If the optional parameter *filter* is given and non-zero, the `exclude_tag` callback is called to remove undesired entries from the context list. To skip the `exclude_tag` callback, either omit the *filter* parameter or set it to zero.

If the optional parameter *alias* is passed the value `1`, `in_context_list` returns the alias names instead of the actual names of the elements allowed in the current context. By default, *alias* is `0`.

index

index(*s1*, *s2*)

This function returns the position in string *s1* of the first occurrence of string *s2*, based at `1`. If the substring is not found, `0` is returned. For example, the value of `index("rnotes9302", "9302")` would be `7`.

indexproc

indexproc (*view_id*, *userule_id*, *userule_parameter*)

This function is useful when used with the `userulehook` on page 969 if you wish to alter the preliminary index before index processing, and/or alter the final index after index processing.

These three arguments to `indexproc` are the same first three values that are passed to the `userulehook` function.

init_done

init_done ()

This function returns `1` (True) if initialization has completed. Only a subset of commands are legal before initialization is finished.

insert

insert(*string*[, *dest*])

This function inserts the string value of the expression *string* into the destination specified by *dest*, which is either a document identifier or a window name. If omitted, the current document is used. It can be one of the window types returned by the `window` function (for example, `edit`, `cmd`, `helpwin1`).

Note

Even if you have applied an alias map to the document, *string* cannot include aliases.

string is interpreted as an SGML-coded string unless the destination window contains an ASCII document-type. Tabs and 8–bit characters are not stripped from text inserted into help windows using this function.

An example of this function is:

```
insert("find", window_doc(" cmd"))
```

This function is similar to the `insert_string -sgml` command except that the *string* argument may be an expression, it is not limited in size, and does not replace any text that might be selected.

This function interprets the processing instruction `<?Pub Caret>` to set the cursor to a specific position within the inserted fragment. The `<?Pub Caret>` must come after a character or tag in the SGML- or XML-coded string. If the `<?Pub Caret>` processing instruction is not included in the string, the cursor is placed according to the following rules:

- If the cursor is in the Edit view, the cursor is placed after the inserted string.
- If the cursor is in the Document Map view, the cursor is placed before the inserted string, though you can control this behavior with the `set docmappastecaret=off` option.

The `insert` function will not do a pending delete regardless of the setting of `set pendingdelete` on page 866.

insert_buffer

`insert_buffer` (*`string`*, *`markup`*[, *`buffer`*[, *`append`*]])

This function inserts the string value of the expression *string* into the paste buffer named by *buffer*, or if not specified, the default paste buffer. If the value of *markup* is not `0`, then the string is interpreted as an XML (or SGML) string. Otherwise no markup is recognized.

If the value of *append* is not specified or is non-zero, the string is appended to the buffer. Otherwise, the string replaces the contents of the buffer.

insert_filep_entity

insert_filep_entity (*name*)

This function provides a reference to an existing file parameter entity (that is, an external parameter entity), which is then used in the current document's internal subset.

name is the name of the entity. A leading percent sign `%` is optional. If the entity has not already been declared, an error message is displayed.

The reference is added in the subset immediately after the declaration of the same entity. For example, if `%myent` has already been declared after the function call `insert_filep_entity("myent")`

the document's SGML file will contain this:

```
<!ENTITY % myent SYSTEM "/usr/myent">
%myent;
```

Note

To ensure the file parameter's declarations are added to the document's declarations, save the document, then choose **Revert to Saved** from the **File** menu.

insert_graphic_file

insert_graphic_file (*path*[, *tagname*[, *doc*]])

This function inserts the graphic tag specified by *tagname* at the cursor and sets the appropriate file or entity reference attribute to the file name specified by *path*. If *tagname* is not specified, a prompt dialog is displayed listing all the graphic tags (as declared in the `.dcf` file) that are valid at the current location. If only one graphic element is valid, then that is used without prompting. If no graphic elements are valid (or are declared in the `.dcf` file), the function returns `0` (False). The function returns `1` (True) if the graphic reference was successfully inserted.

If *doc* is zero or omitted, the current document is used.

The *path* parameter is set using the `oid_set_graphic_pathname` function and stored as a relative path name if possible, as determined by the `graphic_relative_path` function.

insert_string (Function)

insert_string (*string*[, *dest*])

This function inserts the string value of the expression *string* into the destination specified by *dest*, which is either a document identifier or a window name. The window can be one of the window types returned by the `window` function. (For example, `edit` or `cmd`.)

string specifies the text to be inserted at the cursor location. No markup is recognized in the string; the characters are inserted as-is. If *dest* is omitted, the current document is used.

This function is similar to the `insert_string` on page 661 command, except its *string* argument may be an expression and it is not limited in size.

Unlike the `insert` on page 384 function, `insert_string` obeys the setting of the `set pendingdelete` on page 866 option.

insert_pi

insert_pi (*contents*[, *doc*[, *showErr*]])

This function inserts a generic processing instruction with the given *contents*. In Arbortext Editor, a processing instruction is represented with a `_pi` element. If *doc* is not supplied or is `0`, then the current document is used. If *showErr* is supplied and is not `0`, then any errors from the operation are reported.

The function returns `1` if the processing instruction is inserted successfully. Otherwise, `0` is returned.

insert_symbol

ret = **insert_symbol** (*code*[, *font*[, *doc*]])

This function inserts the character specified by *code* into the document specified by *doc* (or the current document if *doc* is `0` or not supplied). If the *font* parameter is supplied, a touchup processing instruction specifying the given font will be inserted if necessary.

The function returns `1` on success and `0` if it fails to insert the character.

insert_tag (Function)

insert_tag (*tagname*[, *dest*[, *noprompt*]])

This function inserts an element into the destination specified by *dest*.

Note

If you have applied an alias map to the document, *tagname* can be an alias, as well as a real name.

- *tagname* — An expression specifying the element to insert.
- *dest* — Optional. A document identifier or a window name. It can be one of the window types returned by the `window` function (for example, `edit`, `cmd`, `helpwin1`). If *dest* is not specified, the current document is used.
- *noprompt* — Optional. If set to a value of `1`, any prompts due to the `promptattrs` on page 876, `requireattrs` on page 879, and `promptgraphicbrowser` on page 876 `set` options are suppressed.

This function returns one (`1`) on success and zero (`0`) on failure.

Example:

```
insert_tag("emphasis", window_doc("edit"),1)
```

inside_tag

`inside_tag` (*`tagname`*`[,` *`doc`*`]`)

This function returns `1` (True) if the cursor is within the tag pair whose start-tag is named *tagname*, regardless of level. If the cursor is not within this tag pair, `0` is returned.

Note

If you have applied an alias map to the document, *tagname* can be an alias, as well as a real name.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

interpreter_addr

interpreter_addr ()

Note

This function has been replaced by the dl_builtin_addr on page 271 function.

The interpreter_addr function returns the address of the Arbortext Command Language interpreter as an integer. This address may be passed to a dynamically loaded function using dl_call to allow such functions to call back into Arbortext Editor to execute an Arbortext Command Language string.

The interpreter takes a single argument, the string to execute, and returns 1 if the command was executed successfully, or 0 if it failed.

Following is an example of some C functions that might be part of a dynamically loaded library.

```
typedef unsigned int (*CMDFUNC) ();
CMDFUNC acl_interp;
void set_acl_addr(void *addr)
{
 acl_interp = (CMDFUNC) addr;
}
int aclcmd(char *str)
{
 if (acl_interp)
 return (*acl_interp)(str);
 fprintf(stderr, "set_acl_addr() must be \
 called first.\n");
 return 0;
}
```

The interpreter address is passed to the library as follows:

```
local set_acl_addr = dl_find(hdll, "set_acl_addr");
dl_call(set_acl_addr, interpreter_addr());
```

is_postscript_printer

is_postscript_printer(*printer*)

This function returns 0 (false) or 1 (true), depending on whether the specified *printer* is a PostScript printer. The printer driver is interrogated to determine whether it renders PostScript output. If the printer is not found, this function returns false.

item_tag

item_tag (*tagname*[, *doc*])

This function returns 1 (true) if the tag named *tagname* is declared as an item element in the .dcf file for the document type associated with *doc*. If *doc* is omitted or 0, the current document is used.

ixkey_charent

ixkey_charent (*oid*)

This is useful if the ixkeycharent hook does some examination and decides it would like to use the built-in processing. The *oid* parameter is the oid of the character entity reference entity name.

java_array_from_acl

java_array_from_acl (*arr*, *descriptor*)

This function converts the ACL array *arr* to a Java object and returns the Java object. The class of the Java object is specified by the *descriptor*.

The *arr* is an ACL array. It can be a normal array or an associative array, depending on the value of the *descriptor*.

The *descriptor* is either a fully-qualified class name, such as java.util.HashMap, or a class descriptor, such as java/util/HashMap or [I. The class descriptor follows Oracle's Java Native Interface Specification. *descriptor* must be the name or the descriptor of one of the following classes:

- A class that implements the java.util.Map interface. For example:

  java.util.HashMap

  java.util.HashTable

  The *arr* must be an ACL associative array. After the conversion, both the key and the value of each element in the newly created Java Map object will be Java String objects.
- A class that implements the java.util.List interface. For example:

  java.util.ArrayList

  java.util.Vector

  The *arr* must be an ACL normal array. After the conversion, each element in the newly created Java List object will be a Java String object.
- A one dimensional primitive type array. The *arr* must be an ACL normal array.

Class descriptors for primitive type arrays are:

boolean	[Z
byte	[B
char	[C
short	[S
int	[I
long	[L
float	[F
double	[D

- A one dimensional `String` array.

  The class descriptor for the Java `String` array is:
  `[Ljava/lang/String;`

This function returns the newly created Java object. It returns 0 if the array conversion failed.

```
local arr[];
# Create an ACL associative array
arr["cereal"] = "breakfast";
arr["steak"] = "dinner";
# Convert the ACL array to a HashMap object
local obj = java_array_from_acl(arr, "java/util/HashMap");
if (obj != 0) {
 # Get the size of this HashMap object
 response("size = " . java_instance(obj, "size"));
 java_delete(obj);
}
```

java_array_to_acl

`java_array_to_acl` (*`obj`*, *`arr`*)

This function converts the Java object *obj* to the ACL array *arr*. All previous elements in the *arr* are discarded.

The *obj* parameter is a Java object. It belongs to one of the following classes:

- A class that implements the `java.util.Map` interface. For example:

  `java.util.HashMap`

  `java.util.HashTable`

  The resulting ACL array *arr* will be an ACL associative array.
- A class that implements the `java.util.List` interface. For example:

java.util.ArrayList

java.util.Vector

The resulting ACL array *arr* will be an ACL normal array.

- A one dimensional primitive type array.

  The resulting ACL array *arr* will be an ACL normal array.

- A one dimensional `String` array.

  The resulting ACL array *arr* will be an ACL normal array.

Each element in the *obj* can be any Java object. Java `String` objects will be directly converted to ACL values. Other Java objects will be turned into Java `String` objects by using the Java `toString` method before converting to ACL values.

This function returns the size of the *arr* after the conversion. It returns -1 if the conversion failed.

```
# Create an ArrayList object
local obj = java_constructor("java.util.ArrayList");
java_instance(obj, "add", "English");
java_instance(obj, "add", "French");
local arr[];
local i;
# Convert the ArrayList object to an ACL array
local size = java_array_to_acl(obj, arr);
java_delete(obj);
for (i = 1; i <= size; i++) {
 response("item " . i . " = " . arr[i]);
}
```

java_console

java_console ([*show*[, *geometry*]])

If the *show* parameter is non-zero or omitted, this function displays the Java Console window. If *show* is `0`, it closes the Java Console.

You can specify the size and position of the Java Console window using the *geometry* parameter. The *geometry* parameter is a string of the form WxH+X+Y, where W and H are the width and height of the window in pixels, and X and Y give the location of the upper left corner of the window. Negative X and Y values give the location of lower right corner of the window with respect to the bottom right corner of the screen. The format of the string allows just the width and height or position to be set by omitting fields. If the Java Console is already displayed, then you can change the size and position of the window by specifying *geometry*.

For example,

```
java_console(1, "500x300-50-50")
java_console(1, "+3+3")
```

```
java_console(0)
```

java_constructor

```
ret = java_constructor(class[, arg1[, arg2[, arg3[,
...arg20]]]])
```

This function creates a Java object by calling *class* and passing it the provided arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *class*. The value returned by the `java_constructor` function is the object created by the called constructor.

java_constructor_modal

```
ret = java_constructor_modal(class[, arg1[, arg2[, arg3[,
...arg20]]]])
```

This function is identical to the `java_constructor` on page 393 function except that Arbortext Editor creates a new thread for the Java call and puts itself in a mode as if a modal dialog box is displayed. The primary advantage to this function is that the Arbortext Editor window will be able to refresh while the Java call is running. The disadvantage of this function is that it is significantly slower than using the `java_constructor` function.

This function creates a Java object by calling *class* and passing it the provided arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *class*. The value returned by the `java_constructor` function is the object created by the called constructor.

java_delete

```
java_delete (javaobject)
```

This function deletes a Java object reference returned by one of the following functions:

- `java_constructor` function on page 393
- `java_constructor_modal` function on page 393
- `java_instance` function on page 394
- `java_instance_modal` function on page 394
- `java_static` function on page 395
- `java_static_modal` function on page 395

Use `java_delete` to remove Java object references that are no longer needed. If you do not delete these references, memory leaks may occur.

```
object = java_constructor("Document", filename)
```

```
count = java_instance(object, "countElements")
# the object is no longer in use, delete it
java_delete(object)
```

java_init

java_init ([*init*])

This function returns 1 (true) if the Java Virtual Machine (JVM) has been initialized. It returns 0 if the JVM has not been started.

If *init* is specified and non-zero, the JVM is initialized, if necessary. In this case, java_init will return 1 if the JVM is initialized successfully, or 0 if it failed to initialize.

For example,

```
inited=java_init()
java_init(1)
```

java_instance

ret = **java_instance** (*object*, *method*[, *arg1*[, *arg2*[, *arg3*[, *...arg20*]]]])

This function executes Java code by calling *method* within *object* and passing it the provided arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *method*. The value returned by the java_instance function is the returned value from the called *method*.

The called *method* must be an instance method. Use the java_static function to call static methods and the java_constructor function to call a constructor. When Java objects are no longer in use, you need to explicitly delete them by using the java_delete

java_instance_modal

ret = **java_instance_modal**(*object*, *method*[, *arg1*[, *arg2*[, *arg3*[, *...arg20*]]]])

This function is identical to the java_instance on page 394 function except that Arbortext Editor creates a new thread for the Java call and puts itself in a mode as if a modal dialog box is displayed. The primary advantage to this function is that the Arbortext Editor window will be able to refresh while the Java call is running. The disadvantage of this function is that it is significantly slower than using the java_instance function.

This function executes Java code by calling *method* within *object* and passing it the provided arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *method*. The value returned by the `java_instance` function is the returned value from the called *method*.

The called *method* must be an instance method. Use the `java_static` function to call static methods and the `java_constructor` function to call a constructor.

java_static

`ret = **java_static** (class, method[, arg1[, arg2[, arg3[, ...arg20]]]])`

This function executes Java code by calling the static method called *method* within *class* and passing the arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *method*. You should supply the package name as part of the class name for the *class* parameter.

The called *method* must be a static method. The value returned by the `java_static` function is the value returned from the called *method*.

Use the `java_instance` function to call instance methods and the `java_constructor` function to call a constructor.

You can use the `java_static` function to call the Java garbage collector by following the example:

```
java_static('java.lang.System','gc');
```

java_static_modal

`ret = **java_static_modal**(class, method[, arg1[, arg2[, arg3[, ...arg20]]]])`

This function is identical to the `java_static` on page 395 function except that Arbortext Editor creates a new thread for the Java call and puts itself in a mode as if a modal dialog box is displayed. The primary advantage to this function is that the Arbortext Editor window will be able to refresh while the Java call is running. The disadvantage of this function is that it is significantly slower than using the `java_static` function.

This function executes Java code by calling the static method called *method* within *class* and passing it the provided arguments *arg1*, *arg2*, *arg3*, and so on. Up to 20 arguments can be passed to the called *method*. The value returned by the `java_static` function is the returned value from the called *method*.

The called *method* must be a static method. Use the `java_instance` function to call instance methods and the `java_constructor` function to call a constructor.

javascript

javascript (*expr*[, *private*])

This function sends the string *expr* to the JavaScript interpreter to be evaluated, returning the result as a string. By default, the JavaScript expression is evaluated in the global shared scope, so previously defined JavaScript functions are accessible. If the *private* parameter is specified and non-zero, the expression is evaluated in a temporary scope. As a result, top-level objects created during evaluation will be discarded on return.

The following example uses an ACL expression to create a corresponding JavaScript expression to evaluate. In this case, it calls the JavaScript `Date.parse` method on the current time and date as returned by the ACL `time_date` function, returning the number of milliseconds between the current time and date and midnight, January 1, 1970 GMT.

```
msecs = javascript('Date.parse("' . time_date() . '")')
```

The following example returns the same result as the previous example using only JavaScript. The `time` ACL function returns a similar result in seconds instead of milliseconds.

```
msecs = javascript('var d = new Date(); d.getTime()')
```

join (Function)

join(*arr*[, *delim*[, *quote*]])

This function concatenates all the elements of the array *arr* and returns the string result. Each element is converted to a string and those strings are concatenated, separated by the specified delimiter string *delim*. If *delim* is omitted, then a comma is used as the separator. If *delim* is a null string, then no separator character is used. If an array element is undefined, for example, *arr* is a sparse array, then a null string is used for the element value so that adjacent delimiters will appear in the string result.

If the *quote* parameter is specified and non-zero, then each element string value is surrounded by double quotes (`"`), and double quote, backslash and newline characters in the value are escaped with a backslash to form a valid ACL string term. For example, the array element value `a "quote"` would be represented in the result string as `"a \"quote\""`, if the *quote* was non-zero.

If *arr* is an associative array, then each element value is prefixed by the element's key followed by a colon. The order of the keys is unspecified.

The following example shows `join` reversing the effect of `split`:

```
local arr[];
split("1 2 3", arr);
local str = join(arr, " ");
# str is "1 2 3"
```

This example shows using `join` for an associative array to create a corresponding JavaScript associative array using object literal syntax:

```
color["red"] = 0xff0000;
color["green"] = 0x00ff00;
color["blue"] = 0x0000ff;
javascript("var color={" . join(color,',',1) . "}")
```

js_source

`js_source` (*`filename`*`[,` *`args`*`[,` *`private`*`]])`

This function reads and executes the JavaScript program from *filename*. The optional *args* parameter is converted from a string to an array and passed to the script in the JavaScript *arguments* global object. The arguments are delimited by blanks and follow the normal quoting conventions on page 75.

If the specified file name does not contain any slashes and is not found in the current directory, the `js_source` function searches the list of directories in the `set loadpath` command on page 848. The `.js` extension is appended to the file name if necessary.

By default, the JavaScript program is evaluated in the global shared scope, so previously defined JavaScript functions are accessible. If the optional parameter *private* is specified and non-zero, the expression is evaluated in a temporary scope, so global shared scope functions are not available. As a result, top-level objects (including function definitions) created during evaluation are discarded on return. The default value is `false`.

Example

```
js_source("deletetags.js")
js_source("jsdoc.js","sample.js",1)
```

jscript

`jscript``(`*`expr`*`[,` *`window`*`])`

This function sends the string *expr* to the Microsoft Windows JScript interpreter to be evaluated, returning the result as a string. If the *window* parameter is not specified, the JScript expression is evaluated in the global JScript script context so previously defined JScript functions are accessible.

The optional *window* parameter is the identifier of a XUI dialog box that has a script context. If *window* is provided, the script executes in that dialog box's script context.

Example

```
name = jscript('Application.prompt("Name", "")')
```

Note

In the Microsoft Script Engine interfaces, “True” is represented by minus one (`1`) and “False” by zero (`0`).

key_cmd

`key_cmd``(`*`keyexpr`*`[, `*`keymap`*`])`

This function returns the command bound to the key specified by *keyexpr* for the keyboard shortcut specified by *keymap*. *keyexpr* is the same as the *keyname* argument to the map on page 669 command (for example, **Control**+**Shift**+**A**). *keymap* is the same as the window argument to `map` and is a window class (for example, `edit`, `cmd`, or `helpwin`), a window name as returned by the `window_name` function, or a user-defined keyboard shortcut denoted by the prefix character `@` (for example, **CTRL**+**S**). If *keymap* is omitted, the keymap associated with the current window is used.

keymap_exists

`keymap_exists` `(`*`name`*`)`

This function returns a value that is not zero if the keymap named *name* exists. This function returns a value of zero the keymap named *name* does not exist. *name* need not begin with the keymap prefix character '@'. If the keymap exists, the function actually returns `1` + the number of references to the keymap, that is, where each window that attaches the keymap is counted as one reference.

languages

`languages` `(`*`arr`*`)`

This function fills the associative array *arr* with the list of language dictionaries which are used by spell checking and the thesaurus. The array is indexed by the language name (see the `set language` on page 845 command) and the values are the language descriptions shown by the user interface. The number of entries in the array is returned.

Example

```
$ret = languages($langs)
```

Returns the following array:

```
$langs["English"]: "English (US)"
$langs["German"]: "German (Deutsch)"
.
```

```
.
.
$langs["Catalan"]: "Catalan (CatalÀ)"
```

legal_name

legal_name (*name*[, *doc*[, *unqualified*]])

This function determines whether a given element or attribute name *name* is a valid XML name. If *name* is a valid XML element or attribute name, this function returns `1` (True). Otherwise, this function returns `0`.

The *doc* argument specifies the identifier of the document to query. If omitted or `0`, the current document is used.

The *unqualified* argument specifies whether namespace qualified names are considered valid for XML documents. If omitted or `0`, then qualified names (a colon separating a valid prefix and a valid name) are considered valid. If `1`, then qualified names are not considered valid.

Note

The XML version for a document is stored in the document's XML declaration processing instruction (PI). That version is used as the basis for testing an XML element or attribute stored in the document.

length

length(*string*)

This function returns the length of *string* taken as a string. For example, the value of `length("rnotes9302")` is `10`.

license

license(*string*)

This function returns a non-zero value if the capability specified by *string* has been licensed at your site.

The following table lists the currently supported strings and the capability each represents.

string	**Capability**
`ImportExport`	Arbortext Import/Export
`ImportWorkbench`	Arbortext Import Workbench
`PrintPublishing`	Print Composer
`WebPublishing`	Web/Wireless Composer
`ChangePage`	Arbortext Change Page for Defense

license_info

`license_info` (*`arr`*)

This function fills the array *arr* with the following elements that give licensing information about the current session>:

- [1] Obsolete – empty string
- [2] Service contract number(s).
- [3] host owning current license
- [4] Obsolete – empty string
- [5] Obsolete – empty string
- [6] Obsolete – empty string
- [7] host name that is running the application
- [8] GUI name, for example, Windows
- [9] time that session started in `time_date` format
- [10] license source
- [11] Obsolete – empty string
- [12] Obsolete – empty string
- [13] number of (physical) processor cores
- [14] number of processor packages (sockets)
- [15] license type – either Floating license or Node-locked license
- [16] host ID – the host ID unique to the system

license_release

`license_release` (*`string`*)

This function releases the license for the product option specified by *string*. It returns `1` (true) if the license specified by *string* was successfully released.

The following table lists the currently supported strings and the product options they represent.

string	**Capability**
`All`	All licensed product options
`ImportExport`	Arbortext Import/Export
`ImportWorkbench`	Arbortext Import Workbench
`PrintPublishing`	Print Composer
`WebPublishing`	Web/Wireless Composer
`ChangePage`	Arbortext Change Page for Defense

linenum

`linenum``([`*`doc`*`])`

This function applies the sample line numbering application found in the `samples` folder of your installation directory. It looks for `atipl` markup in the document and specifies a set of generic attributes, clearing all others. In particular, it sets the `attr1` attribute of `atipl:startpage` tag to the value of the `number` attribute. It also the sets the `attr1` attribute of `atipl:startline` tags to the current line number. The line number is reset with every start page.

The *doc* parameter specifies the document to modify. The current document is assumed if this parameter is not specified.

If the operation fails the function returns `0`. If the operation succeeds then the function returns `1`.

For more detailed information on the page layout elements and their attributes, see http://www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

link_idref_attr_name

`link_idref_attr_name` `(`*`tagname`*`[, `*`doc`*`])`

This function returns the name of the IDREF attribute (for example, “linkend”) for the link tag named *tagname*. If no such attribute is defined for the tag, `link_idref_attr_name` returns the null string. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name. However, this function will return an attribute's real name, not its alias.

The IDREF attribute is used by the **Insert Link** dialog box for document (internal) links.

link_tag

link_tag (*tagname*[, *doc*])

This function returns 1 (true) if the tag named *tagname* is declared as a link element in the .dcf file for the document type associated with *doc*. If *doc* is omitted or 0, the current document is used.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

Note

If the document type definition for your document instance includes the NAMECASE GENERAL NO setting, then the *tagname* parameter will be case-sensitive.

link_tag_name

link_tag_name ([*doc*])

This function returns the name of the primary link element for the document type associated with *doc*. It returns the null string if no link tag was designated for the document type or if *doc* is invalid or an ASCII file. If *doc* is omitted or 0, the current document is used.

Note

Even if you have applied an alias map to the document, this function will return an element's real name, not its alias.

link_uri_attr_name

link_uri_attr_name (*tagname*[, *doc*])

This function returns the name of the Universal Resource Indicator (URI) attribute (for example, “url”) for the link tag named *tagname*. If no such attribute is defined for the tag, `link_uri_attr_name` returns the null string.

Note

Even if you have applied an alias map to the document, this function will return an attribute's real name, not its alias.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

link_valid

`link_valid` `([`*`back`*`[, `*`doc`*`]])`

This function determines whether a link operation is valid at the current location in the document specified by *doc*.

The following arguments are optional:

back — if not supplied or 0, the function will test the validity of returning to the position before the previous link operation. If non-zero, it will test whether there is a link to be followed at the current location.

doc — if not supplied or 0, the current document is used. Specify an alternative document with this argument if required.

The function returns 1 if a valid link operation is possible, otherwise it returns 0.

list_response

`list_response` `(`*`arr`*`[, `*`title`*`[, `*`label`*`[, `*`default`*`[, `*`verify`*`[, `*`help`*`]]]]])`

This function displays a scrolling list dialog panel and returns the user's response. The null string is returned if the user selected **Cancel** or dismissed the panel without selecting anything. The array of choices is specified by the argument *arr*, which must be the name of a normal array. That is, the choices are obtained from:

arr `[1]..` *arr* `[count(` *arr* `)]`

The remaining arguments may all be expressions and are optional:

title specifies the window manager title for the panel.

label specifies the label displayed at the top of the scrolling list panel.

default gives the default choice and is displayed in the input field; if not specified, there is no default.

If *verify* is non-zero or not specified, then the user's response is restricted to the array of choices.

help is the text displayed if the user presses the **Help** button on the panel. If not specified, then the **Help** button is insensitive.

list_stylesheets

`list_stylesheets` `()`

The `list_stylesheets` function returns the paths to all stylesheets in the compiled stylesheet cache. Arbortext Editor stores XSL stylesheets in this cache after they have been compiled during a publishing process. By storing these stylesheets, Arbortext Editor avoids having to recompile them during subsequent publishing runs.

You can use a path returned by this function as an argument to the `clear_stylesheet` function.

list_tag

`list_tag` (*`tagname`*`[,` *`doc`*`]`)

This function returns `1` (true) if the tag named *tagname* is declared as a list element in the `.dcf` file for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

locale_file_name

`path=`**`locale_file_name`**(*`file`*`[,` *`dir`*`]`)

This function returns the location of the localized file specified by the *file* parameter (remember to include the extension). The optional *dir* parameter is available for manually specifying a directory for searching. If, for example, you specified a *dir* of `/test`, the function would search for the specified *file* in the following locations:

1. `\test\lib\locale\`*`syslocale`*
2. `\test\locale\`*`syslocale`*

3. `\test\lib`
4. `\test`

The *syslocale* subdirectory corresponds to the current system locale setting. You can determine the current system locale using the `getlocale` function. If the three-character locale directory is not found (for example, "ENU"), Arbortext Editor tries the two-character version formed by dropping the last letter (for example, "EN").

If the optional *dir* parameter is not specified, the function automatically substitutes the Arbortext Editor installation directory (*`Arbortext-path`*) for the *dir* parameter and searches the same set of subdirectories.

If the function executes successfully, it returns the string for the path to the desired *file*. If the function fails, it returns the value *`Arbortext-path`*`\lib\`*`file`*, where*`Arbortext-path`* is the Arbortext Editor installation directory, and *file* is the value of the *file* argument.

Examples

```
path = locale_file_name('message.amo')
path = locale_file_name('message.amo','/test')
```

looking_at

`looking_at` (*`regexp`*`[,` *`doc`*`]`)

This function returns `1` (True) if the text following the cursor matches the regular expression *regexp*. The regular expression is compiled with tag scanning enabled (that is, as if `set tagscan=on`). Thus, `looking_at("</title>")` returns `1` (True) if the cursor is positioned before an end `title` tag.

Note

Even if you have applied an alias map to the document, *regexp* cannot include aliases.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Because double quoted strings in expressions get interpreted for backslash escape sequences when the command is parsed, if you wish for a character not to be interpreted as a regular expression meta character, you must double the `\`. For example, to match the literal character <, you would specify:

```
looking_at("\\<")
```

Alternatively, single quotes may be used to suppress backslash interpretation, making the following statement equivalent to the previous one.

```
looking_at('\<')
```

The `set case` option determines whether the case of letters is significant when performing the match. For example, if `set case=off`, then `looking_at("[aB]")`

would match `A` or `b` as well as `a` or `B`.

lookup

`lookup``(`*`word`*`, `*`definitions[]`*`)`

This function retrieves the definitions for the specified *word*. If *word* is empty or contains spaces, or markup is included or is not found, this function returns 0. Otherwise, the number of definitions found is returned.

definitions is a numeric indexed array containing the word part of speech and definition in the form `<part-of-speech>:<definition>`.

lookup_replacements

`lookup_replacements` `(`*`index`*`, `*`replacements[]`*`, `*`typeFlag = 0x01`*`)`

This function retrieves the possible replacements for the definition of the word specified in the previous call to the `lookup` function. If there was no previous call or the word was not found, then 0 is returned. Otherwise the number of replacements is returned.

index is the numeric index for the definition returned in the previous call to the `lookup` function or 0 if a list of suggestions is requested.

replacements is a numeric indexed array containing the possible replacement words.

typeFlag is one of the following flags to specify the type of replacement:

- `0x01` — Synonym
- `0x02` — Compared
- `0x04` — Related
- `0x08` — Contrast
- `0x10` — Antonym

index is the numeric index for the definition returned in the previous call to the `lookup` function.

lookup_types

`lookup_types` `(`*`index`*`)`

This function returns a bit mask indicating the possible word types for the definition index of the word specified in the previous call to the `lookup` function. If there was no previous call or the word was not found, then 0 is returned. Possible flags are:

- `0x01` - Synonym
- `0x02` - Compared
- `0x04` - Related
- `0x08` - Contrast
- `0x10` - Antonym

index is the numeric index for the definition returned in the previous call to the `lookup` function.

low_bound

`low_bound` (*`arr`*)

This function returns the minimum subscript allowed for the array *arr*. If the array was not declared a fixed size with the `local` or `global` statement, then `low_bound` returns the lowest subscript used or 1 if the array is empty. If *arr* is an associative array, the function returns a null string.

macro_exists

`macro_exists` (*`name`*[`,` *`doc`*])

This function returns `1` (`True`) if the macro specified by *name* is defined in the scope specified by the document *doc*. If *doc* is omitted or `0`, the current document is used. Predefined commands and aliases are not recognized as macros in this context.

- *doc* — The identifier of the document tree.

This function also returns `1` if no arguments are supplied and one or more macros are defined.

macro_pause_recording

`macro_pause_recording`([*`resume`*])

This function pauses recording if a macro is being recorded. If the optional parameter *resume* is given and non-zero, then macro recording is resumed if it was paused.

If the **Macros** toolbar is displayed, this function changes the state of the corresponding toggle button.

macro_record

macro_record (*name*[, *scope*[, *desc*[, *key*[, *doc*]]]])

This function starts a macro recording session on the specified document.

- *name* — The name of the macro. It must consist of valid XML name characters. If a macro named *name* already exists in the specified scope, the new macro will replace it. Do not use an ACL command name as the name of a macro. Using an ACL command name as a macro name may cause unexpected results when running the macro or ACL command.
- *scope* — Specifies the target macro scope which determines which macro file is updated or created. *scope* and is one of the following values:
  - 1 — User scope. The macro file is user.mcf and is stored in the application data directory. The application data directory is typically located at C:\Documents and Settings*username*\Application Data\PTC\Arbortext\Editor.
  - 2 — Document-type scope. The macro file is *doctype*.mcf, where *doctype* is the base name of the document type directory associated with *doc*. The macro file is written to the document-type directory (which must have write access).
  - 3 — Document scope. The macro file is *docname*.mcf, where *docname* is the base name of the document specified by *doc*.

  If *scope* is omitted, then user scope is assumed.
- *desc* — Optional. A description of the macro to be displayed in the **Macros** dialog box.
- *key* — Optional. A key binding for the macro. It has the same form as the *keyname* argument to the map on page 669 command.
- *doc* — Optional. The identifier of the document on which the macro recording session is started. If *doc* is omitted or 0, the current document is used.

macro_record function displays the **Macros** toolbar and changes the window cursor to indicate that mouse operations are ignored while recording. Macro recording continues until the macro_stop_recording on page 410 function is called.

macro_record returns 1 (True) if recording was started or 0 (False) if a macro is already being recorded.

macro_record_cmd

macro_record_cmd (*command*[, *doc*])

If a macro is being recorded on the document specified by *doc*, this function assigns the string specified by *command* as the command to be recorded for the current event.

- *command* — The string to assign as the command recorded for the current event.

  command replaces any command that would have been recorded for the event; for this reason this function should be called after any changes are made to the document.

  The command string to record should not specify an explicit document or window id if the macro will be used in another session. Most built-in functions use the current document if the document parameter is omitted. If a function requires a document or window parameter, `current_doc` or `current_window` should be specified.
- *doc* — Optional. The identifier of the document on which the macro recording session is started. If *doc* is omitted or `0`, the current document is used.

macro_recording

`macro_recording` (`[`*`doc`*`]`)

If *doc* is not specified, this function returns `1` (`True`) if a macro is currently being recorded on any document.

If *doc* is specified, `macro_recording` returns `1` if macro recording was started on the document specified by *doc*. If a macro is being recorded on another document, `macro_recording` returns `0`. If *doc* is `0`, the current document is used.

- *doc* — Optional. The identifier of the document tree.

macro_run

`macro_run` (*`name`*`[, `*`doc`*`]`)

This function runs the macro named *name* in the document-scope specified by *doc*.

This function returns `1` (`True`) if the macro was executed. `macro_run` returns `0` if no such macro exists in the current scope. `macro_run` can not be used to execute command aliases.

- *name* — The macro to run.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or `0`, the current document scope is used.

If *doc* is the current document, the macro can also be run by giving its name. That is:

```
macro_run("MyMacro")
```

is the same as

```
MyMacro
```

macro_running

macro_running ()

This function returns non-zero if a macro is running.

macro_stop_recording

macro_stop_recording ([*cancel*])

This function finishes recording if a macro is being recorded. If the optional parameter *cancel* is given and non-zero, then macro recording is aborted. Otherwise, the recorded macro is saved as specified by the parameters to the corresponding macro_record on page 408 function.

If the **Macros** toolbar is displayed, this function hides it.

marked_section_tag

marked_section_tag (*tagname*[, *doc*])

This function returns 1 (True) if the tag specified by the entity *tagname* is an SGML marked section declaration. The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

Note

If the document type definition for your document instance includes the NAMECASE GENERAL NO setting, then the *tagname* parameter will be case-sensitive.

marking

marking([*doc*])

This function determines whether a selection operation is in progress for the given document *doc*. If *doc* is omitted or set to zero, the current document is used.

The function returns `1` if a selection operation is in progress. Otherwise, the function returns `0`.

match

`match``(`*`string`*`, `*`regexp`*`)`

This function searches *string* for the first substring matching *regexp*. *regexp* is a regular expression, the same as for the caret on page 621 commands or `looking_at` function except that tag delimiters < and > are not meta-characters. `match` returns an index of start of match, if match succeeds, or `0` if the match fails.

match_length

`match_length` `(`*`n`*`)`

This function returns the length of the *n*th subexpression matched from the previous call to `match`. If *n* is `0`, then the length of the entire matched string is returned.

match_result

`match_result` `(`*`n`*`)`

This function returns *n*th subexpression matched from previous call to `match`. If `0`, match_result returns entire matched string. This is equivalent to: `substr(` *`string`*`, match_start(` *`n`*`), match_length(` *`n`*`))` where *string* is the first argument to the previous call to `match`.

match_start

`match_start` `(`*`n`*`)`

This function returns the index of the first character of the *n*th subexpression matched from the previous call to `match`. If *n* is `0`, then the index of the first character matched is returned.

max

`max``(`*`n1`*`, `*`n2`*`)`

This function returns the maximum of numbers *n1* and *n2*.

mblen

mblen(*bytestr*[, *charset*])

This function returns the number of bytes in *bytestr* that form whole characters in the character set specified by *charset*, which may be a single-byte character set (SBCS), multi-byte character set (MBCS), or double-byte character set (DBCS). The length returned should be passed to `mbstoucs` to convert the string to Unicode. *charset* is one of the following strings:

- iso8859-1
- iso8859-2
- iso8859-3
- iso8859-4
- iso8859-5
- iso8859-7
- iso8859-8
- iso8859-9
- ps_ascii
- ps_symbol
- jeuc
- sjis
- big5hku
- gb2312
- koi8
- ksc5601
- utf-8
- Unicode

If *charset* is not specified, the system character set is assumed.

mbstoucs

mbstoucs (*bytestr*, *len*[, *charset*])

This function converts the byte string *bytestr* encoded in the character set charset to a Unicode string. *len* specifies the number of bytes in *bytestr* to examine, and is normally the return value from a call to `mblen`.

charset is one of the character sets listed in the description of `mblen`. If *charset* is not specified, the system character set is assumed.

If the character set is a multi-byte character set, then it is possible that the last few bytes in *bytestr* do not form a whole character but are the prefix of a multi-byte character. In this case, they should be joined with the rest of the character for a subsequent call to `mbstoucs`. See the following example.

Example

This example shows how to convert a Japanese file encoded in JEUC into SJIS. For brevity, no error checking is done. Also, this example would need additional code to handle converting from or to 16-bit character sets, for example, to handle byte-swapping or the Unicode text file signature.

```
inf = open(jeucfile, "rb")
outf = open(sjisfile, "wb")
while ((len = read(inf, buf, 512)) 0)
 {
# append what we just read to any left over bytes
# from previous read
 bstr = remb . buf;
# compute how many bytes form whole characters
 mb_len = mblen(bstr, "jeuc");
# and copy the remainder to REMB
 remb = substr(bstr, mb_len+1);
# convert to Unicode
 ucsstr = mbstoucs(bstr, mb_len, from_charset);
# and out to SJIS
 mbstr = ucstombs(ucsstr, "sjis");
 write(outf, mbstr);
}
close(inf);
close(outf);
```

menu_active

ret = **menu_active** (*menupath*[, *active*])

This function returns the enabled state (`1` or `0`) of the menu item specified by *menupath*. If the parameter *active* is supplied, the menu item is enabled or disabled based on the value of *active* and the function returns the previous state of the menu item.

The function returns `-1` if *menupath* does not specify an existing menu item.

menu_checked

menu_checked (*menupath*)

This function returns `1` (True) if the menu item specified by *menupath* is checked. It returns `0` if the menu item is not checked or *menupath* does not specify a toggle-style item.

The *menupath* is a string that defines the path to the menu. Refer to the related topics for more information on menu paths.

menu_cmd

menu_cmd (*menupath*)

This function returns the command bound to the menu item specified by *menupath*. If there is no such menu, it returns the null string. If the menu item specified by *menupath* is not a command type item, it returns one of the following keywords:

- `#title` — item is a menu title
- `#menu` — item is a pullright menu
- `#nop` — item is not-selectable
- `#help` — item has only help text associated with it.

The *menupath* is a string that defines the path to the menu. Refer to the related topics for more information on menu paths.

Examples

```
menu_cmd(".File.Print")
menu_cmd(":EditPopup.Paste")
menu_cmd(".Options.Full Menus")
menu_cmd(".File.Preview.All Passes")
```

menu_exists

menu_exists (*menupath*)

This function returns `1` (True) if the menu or item specified by *menupath* exists.

The *menupath* is a string that defines the path to the menu. Refer to the related topics for more information on menu paths.

menu_item_array

menu_item_array (*menupath*, *arr*)

This function fills the array *arr* with all the menu item labels in the menu specified by *menupath*, returning the number of items. Menu titles are not returned.

Each menu item label is prefixed by the name of the menu to form a menu path for use in other menu functions or commands as shown in the following example.

Menu separators are returned using the form *menuname*.#*N*, where *N* specifies the position in the menu. For example, Insert.#10 specifies a menu separator which is the tenth item on the Insert menu.

The *menupath* is a string that defines the path to the menu. As a special case, "." refers to the menu bar for the current window and all top level menu items are returned.

The function returns the number of labels stored in the array.

Example:

```
function show_menu_items(menu)
{
 local items[];
 local cnt = menu_item_array(menu, items);
 local i;
 for (i = 1; i <= cnt; i++) {
 local label = items[i];
 local cmd = menu_cmd(label);
 eval label, "\t", cmd output=>*
 if (cmd == "#menu") {
 show_menu_items(label);
}
}
}
# Recursively enumerate the menu items and commands for the Insert menu
show_menu_items(".Insert")
```

menu_item_count

menu_item_count (*menupath*)

This function returns the number of items in the menu specified by *menupath*. Returns −1 if *menupath* does not specify an existing menu. If *menupath* specifies an item, that is, not pulldown menu, the number of items in the containing menu is returned. Menu titles do not count as items.

The *menupath* is a string that defines the path to the menu. Refer to the related topics for more information on menu paths.

menu_popup

menu_popup (*menu*[, *window*[, *geometry*]])

This function posts the shortcut menu named *menu* in the window containing the current document or in a XUI dialog box. menu_popup returns 1 if the menu was posted, or 0 if no such shortcut menu exists. This function can be mapped to a button or key press. Typically, the right mouse button is used to post shortcut menus.

- *menu* — The name of a shortcut menu defined within a menu configuration file. If *window* is specified and is the window ID of a XUI dialog box, *menu* may also be the XUI ID of a context or dropdown menu defined within the dialog box. Any ACL event dispatched for this menu can be processed by adding an ACL event callback. For more details, refer to the XUI documentation in the *Customizer's Guide*.
- *window* — Optional. Specifies the ID of the parent window.
- *geometry* — Optional. Specifies the location for posting the menu. The value of *geometry* takes the form `+X+Y`

  For example:
  `+200+100`

  Where *X* is the horizontal screen coordinate of the upper-left corner of the control, and *Y* is the vertical screen coordinate of the upper-left corner of the control.

message_box

message_box (*message*, *flags*[, *title*])

This function displays a message box with the text *message* and optional title *title*. The *flags* parameter determines what predefined buttons and icons display in the message box, and is formed by ORing the flags from the following groups of flag bits.

Specify one of the following flags to indicate the buttons that will display in the message box:

- 0x00 — Display OK button only. This is the default.
- 0x01 — Display OK and Cancel buttons.
- 0x02 — Display Abort, Retry, and Ignore buttons.
- 0x03 — Display Yes, No, and Cancel buttons.
- 0x04 — Display Yes and No buttons.
- 0x05 — Display Retry and Cancel buttons.

Specify one of the following flags to indicate the icon to display in the message box. If you do not specify one of these flags, an icon does not display.

- 0x10 — Display the Error (Stop) icon. This icon is typically used with the Abort, Retry, and Ignore buttons.
- 0x20 — Display the Question icon. This icon is typically used with the Yes and No buttons.
- 0x30 — Display the Warning icon.
- 0x40 — Display the Information icon.

Specify one of the following flags to indicate the default button:

- 0x000 — The first button is the default. This is the default if no other default button flag is specified.
- 0x100 — The second button is the default.
- 0x200 — The third button is the default.

If *title* is not specified or is a null string, the default title `$progname Message` is used.

The return value is one of the following:

- `1` — The first button (Yes, OK, Abort, or Retry) was pressed.
- `2` — The No button was pressed.
- `3` — The third button (Cancel or Ignore) was pressed.

If the dialog box has a **Cancel** or **Ignore** button, the function returns `3` if the **Cancel** or **Ignore** button or **ESC** key was pressed, or if the dialog box was closed from the **Close** system menu or **Close** button. If the dialog box does not have a **Cancel** or **Ignore** button and is closed with the **ESC** key or by the **Close** system menu or **Close** button, the function returns `2` if the dialog has only **Yes** and **No** buttons. If the dialog box only has an **OK** button, it returns a `1`.

Examples

```
message_box("Save new file?", 0x03+0x20)
message_box("Document is incomplete.", 0x40, "Incomplete")
```

min

min(*n1*, *n2*)

This function returns the minimum of the two numbers *n1* and *n2*.

modified

modified ([*doc*[, *value*]])

This function returns `1` (True) if current document has been modified (otherwise, zero). The *doc* parameter specifies the identifier of the document tree to query. If omitted or `0`, the current document is used. The *value* parameter specifies the new value for the “modified” state of *doc*. Note that the change to *value* takes place after determining the return value.

modify_attr

modify_attr (*attrname*, *value*[, *doc*])

This function sets the attribute *attrname* to *value* for the element to the left of the cursor in the document specified by *doc*. If *doc* is omitted or `0`, the current document is used.

If successful, `modify_attr` returns `1` (`True`). If *attrname* doesn't exist for the element, or the document is read-only, `modify_attr` returns `0`.

`modify_attr` can be used to assign XML namespace attributes to elements in an XML document.

The function call:
```
modify_attr(name, value, doc)
```
is the same as:
```
oid_modify_attr(oid_current_tag(doc), name, value)
```

modify_file_entity

`modify_file_entity` (*`name`*`[,` *`sysid`*`[,` *`pubid`*`[,` *`type`*`[,` *`doc`*`]]]])`

This function allows you to modify a file entity declaration.

name is the name of an existing file entity. The leading `&` is optional.

sysid is the new system identifier value (if there is one).

pubid is the new public identifier value (if there is one).

type is either empty, `normal`, or `SUBDOC` to declare the entity as a SUBDOC entity.

doc is the document identifier. By default, the current document is assumed.

modify_graphic_entity

`modify_graphic_entity``(`*`name`*`,` *`notn`*`[,` *`sysid`*`[,` *`pubid`*`]])`

This function allows you to modify a graphic entity declaration.

name is the name of an existing graphic entity.

notn is the name of the entity's new notation. The leading `&` is optional.

sysid is the new system identifier value (if there is one).

pubid is the new public identifier value (if there is one).

modify_text_entity

`modify_text_entity` (*`name`*`,` *`text`*`[,` *`type`*`])`

This function allows you to change the declaration of a text entity.

name is the name of an existing text entity. The `&` prefix is optional.

text is the new replacement text.

type is supplied, and should be one of the following:

- CDATA — character data
- PI — processing instruction
- MS — marked section
- MD — markup declaration

If omitted, the entity is a normal entity (that is, the markup within the replacement text is recognized).

mouse_at

mouse_at ([*window*[, *flags*]])

This function returns a string describing the object beneath the mouse cursor in the window specified by *window*. If the argument is omitted or `0`, the current window is used.

The following strings are returned by `mouse_at`:

- `attr.`*`Name`* — in the Document Map, the object is an attribute line and *Name* is the attribute name displayed, for example, `attr.id`.
- `attrval.`*`Name`* — in the Document Map (or an expanded attribute in Column view), the object is an attribute value and *Name* is the attribute name for which the value is displayed, for example, `attrval.id`.
- `cellattrval.`*`Name`* — over a cell in Column view, and *Name* is the attribute name for which the value is displayed in the cell, for example, `cellattrval.id`. If no attribute exists for the element in a Column view cell, then *Name* is set to `unknown`.
- `columnview.ruler_boundary`— in Column view, the object is a boundary between two columns in the heading (used for column resizing).
- `equation`— object is an equation image.
- `icon.`*`icontype`* — object is the display icon *icontype* for the current element. Possible return values for *icontype* are in the following table.

Available icon names

Name	Icon	Default use
`Attribute`		Indicates that an element has attributes assigned to it.
`BadAttribute`		Indicates that an element has attributes assigned to it and at least

Available icon names (continued)

Name	Icon	Default use
		one of those attributes is not legal for the current document type definition.
BadElement		Indicates an element whose element name is not legal for the current document type definition.
CheckedOut		Indicates a document object accessed by a Repository Adapter is locked or checked out by the current user.
Comment		Indicates a comment element.
Contracted		Indicates that the element's contents are collapsed and hidden from view.
DataMarkedSection		Indicates that an SGML element is part of a data Marked Section
DocObject		Indicates a document object accessed by a Repository Adapter that is neither locked nor checked out.
Document		Indicates a document.
Element		Indicates an SGML or XML element.
Empty		Indicates an element with no content.
End		Indicates an end tag.
Equation		Indicates an equation element.
Expanded		Indicates that the element's contents are expanded for viewing.
FileEntity		Indicates a referenced file entity.
Graphic		Indicates a graphic element.
GrayCheckedOut		Indicates a document object accessed from a Repository Adapter that is locked or checked out by the current user but is unavailable.
GrayDocObject		Indicates a document object accessed from a Repository Adapter that is neither locked nor checked out by the current user and is unavailable.
GrayLocked		Indicates a document object accessed from a Repository Adapter that is locked or checked out by another user and is unavailable.

Available icon names (continued)

Name	Icon	Default use
`IgnoreMarkedSection`		Indicates that an SGML element is part of an ignored Marked Section
`Locked`		Indicates a document object accessed by a Repository Adapter is locked or checked out by another user.
`Missing`		Indicates where a required element is missing.
`None`		No icon displayed.
`ReadOnly`		Indicates a element that is not editable.
`Table`		Indicates a table element.

- `icon.ElementInlineContent`— in Column view, the object is an element with content (text or inline elements) that is not displayed.
- `name.`*`TagName`* — in the Document Map, the object is the name field next to the icon and *TagName* is the name displayed, for example, `"name.para"` if over the paragraph element `<para>`.
- `null` — no object, in the case of an empty file.
- `tag` — object is a tag (either start or end). This includes entity references, processing instructions, and all other non-image markup.
- `tabl.`*`location`* — the object is the location pertaining to the current table. Possible return values for *location* are:
  - `toprule` is the top most rule for a table.
  - `bottomrule` is the bottom most rule for a table.
  - `rule_horiz` is any horizontal rule of a table besides the top and bottom rules.
  - `rowmargin` is the left-most rule of a table, or to the left of it.
  - `rightrule` is the right-most rule of a table, or to the right of it.
  - `rule_vert` is any vertical rule of a table besides the right and left-most rules.
  - `cellmargin` is in the left margin of a cell. This is used for cell selecting.
  - `cell` is any other location in a cell.
  - `ruler_horiz` is any horizontal rule of the table ruler.
  - `ruler_vert` is any vertical rule of the table ruler.

- `other` is reserved for any other location in a table.

- `text` — object is text.

flags is an optional bitmask that controls the value returned when the cursor is over a tag in a table cell. It has the following flag:

- `0x001` — Being over a table cell is more significant than being over the cell's contents. For example, if the cursor is over a tag that is inside a table cell, the function returns `tabl.cell` when this bit is set. Otherwise, the function returns `tag`.

Note, that this function is most useful when called from a mouse button mapping. If invoked outside a mouse button mapping, then the previous mouse pointer position in the specified window is used.

mouse_in_selection

`mouse_in_selection` `([`*`doc`*`])`

This function returns `1` (True) if the mouse pointer is within the current selection of the document specified by *doc*, or the current document if *doc* is omitted. Returns `0` (False) if the pointer is not within the selection, or if there is no selection in the document identified by *doc*.

mouse_set_waiting

`mouse_set_waiting` `()`

This function changes the mouse cursor to the standard waiting cursor to indicate that the application is blocked. The cursor will be restored to the normal one after the current event has completed.

msp_entity_names

`msp_entity_names` `(`*`arr`*`[, `*`doc`*`])`

The `msp_entity_names` function fills the array *arr* with an alphabetical list of marked section parameter entity names which are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

Only those marked section parameter entities declared in the declaration subset are included in the array *arr*. Parameter entities declared in the DTD are not included.

notation_exists

notation_exists (*name*[, *doc*])

The notation_exists function returns 1 (True) if the notation named *name* is a valid notation in the current document. If the notation is not declared, 0 is returned.

The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

notation_names

notation_names (*arr*[, *doc*[, *exclude*]])

The notation_names function fills the array *arr* with an alphabetical list of notation names which are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

If the *exclude* argument is 1, then the excludegraphicnotation hook is called to potentially exclude any notations from the array. If omitted or 0, all notations are included.

Note

Notations declared in the DTD are included in the array *arr* in addition to those declared in the declaration subset.

notation_parfile

notation_parfile (*notation*[, *doc*])

This function returns the name of the parameter file entity in which the notation was declared (if the entity was declared in a notation file entity). Otherwise, the null string is returned.

notation is the name of the entity.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

notation_pubid

`notation_pubid` (*`name`*`[,` *`doc`*`]`)

The `notation_pubid` function returns the PUBLIC identifier from the declaration for the notation *name*. It returns the null string if the notation does not exist or if no PUBLIC identifier was specified.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

notation_source

`notation_source` (*`name`*`[,` *`doc`*`]`)

The `notation_source` function returns a number indicating the origin of the declaration for the notation named *name*. The return value is one of the integers:

- `-1` — notation not declared.
- `0` — notation defaulted.
- `1` — notation declared in declaration subset (private markup) only.
- `2` — notation declared in DTD only.
- `3` — notation declared in both DTD and declaration subset.

The argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

notation_sysid

`notation_sysid` (*`name`*`[,` *`doc`*`]`)

The `notation_sysid` function returns the SYSTEM identifier from the declaration for the notation *name*. It returns the null string if the notation does not exist or if no SYSTEM identifier was specified.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

ns_schema_validate_batch

ns_schema_validate_batch(*ns_schema[]*[, *schema*[, *doc*[, *flag*]]])

Validates a document against a set of schema definitions, including non-namespaced and namespaced schema definitions.

- *ns_schema[]* — An array of path and file names for namespaced schema definition (`XSD`) files.
- *schema* — Path and file name of a non-namespaced schema definition (`XSD`) file. Defaults to empty if not supplied.
- *doc* — The document to validate against the schemas. If not specified, the current document is used.
- *flag* — Enables you to control how the *doc* is validated against the *schema*. The following flags are available:
  - `1` — Check identity constraints.
  - `2` — Check entities.
  - `4` — Check the document structure, including content and data types.
  - `8` — Check attribute values.
  - `16` — Report errors.
  - `32` — Define how to report errors.

    If set, either a list of errors or a message saying that there are no errors is written to an event log document. The event log can be retrieved by calling the ACL function `get_schema_log_doc()`, which will return the document ID as an integer. Use the document ID to inspect the content of the document, write it to disk, display it, or carry out manipulations as with an open document.

    If not set, any schema errors are reported in the same way as errors reported by a completeness check. Errors are reported in a popup **Parser Messages** window. If there are no errors, no window is displayed.
  - `64` — Fall back to the schema associated with *doc*, if the specified *schema* is not available or does not contain element definitions that occur in *doc*.

  By default, the function checks all of these areas and reports errors (not to the buffer). If you only want to validate one or more of these areas, set the appropriate flag or flags. Add the flag values to set multiple flags. For example, to just check identity constraints and the document structure, set *flag* to `5` (`1` + `4`).

The function returns one of the following values:

- `0` — The validation failed.
- `1` — *doc* is valid.

All `ns_schema_validate_batch` results are shown in the event-log document. Refer to `schema_validate` on page 508 for similar functionality.

numbered_list_block_tag_name

`numbered_list_block_tag_name``(`*`arr`*`[, `*`doc`*`])`

This function returns the name of the primary numbered list block tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted when users select the numbered list button on the Application toolbar.

If a tag name is returned, `numbered_list_block_tag_name` function returns *arr* with an attribute name and attribute value, if they were specified for the tag in the `.dcf` file.

The function returns the null string if there isn't a numbered list block tag designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

For example,

```
local tag, arr[];
local attr, val;
tag = numbered_list_block_tag_name(arr);
attr = arr['attribute'];
val = arr['attribute_value'];
```

numbered_list_block_tag_name_ns

`numbered_list_block_tag_name_ns``(`*`arr`*`[, `*`doc`*`])`

This function returns the namespace URI prefixed to the primary numbered list block tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted when users select the numbered list button on the Application toolbar.

If a tag name is returned, `numbered_list_block_tag_name` function returns *arr*.

The function returns the null string if there isn't a namespace prefix, if no numbered list block tag is designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

numbered_list_item_tag_name

`numbered_list_item_tag_name``(`*`arr`*`[, `*`doc`*`])`

This function returns the name of the primary numbered list item tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted, along with the numbered list block tag, when users select the numbered list button on the Application toolbar.

If a tag name is returned, `numbered_list_item_tag_name` function returns *arr* with an attribute name and attribute value, if they were specified for the tag in the `.dcf` file.

The function returns the null string if there isn't a numbered list item tag designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

For example,

```
local tag, arr[];
local attr, val;
tag = numbered_list_item_tag_name(arr);
attr = arr['attribute'];
val = arr['attribute_value'];
```

numbered_list_item_tag_name_ns

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

This function returns the namespace URI prefixed to the primary numbered list item tag specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. This tag is inserted, along with the numbered list block tag, when users select the numbered list button on the Application toolbar.

If a tag name is returned, `numbered_list_item_tag_name_ns` function returns *arr*.

The function returns the null string if there isn't a namespace prefix, if no numbered list item tag is designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

oct

oct(*value*)

This function returns the decimal value of *value* interpreted as an octal string. For example, the value of `oct("10")` is `8`.

oid_asis

oid_asis ([*oid*])

This function returns whether or not the contents of *oid* constitutes an “asis” region. The function returns a `1` if the oid is an “asis” region or a `0` if the oid is not an “asis” region. If the *oid* argument is omitted, then the current oid of the current document is assumed.

An “asis” region is one where newlines in the document content are preserved and cause line breaks both in the Editor pane and in published output. A region is “asis” if the FOSI specified justification (that is, quadding) in effect is “asis”, or if the document is an untagged ASCII file.

oid_attr

`oid_attr` (*`oid`*, *`attrname`*)

This function returns the value of the attribute *attrname* for the specified object *oid*, or the null string if the element identified by *oid* has no such attribute.

> **Note**
>
> If you have applied an alias map to the document, *attrname* can be either an alias or a real name. However, this function will return an attribute value's real name, not its alias.

oid_attr_list

`oid_attr_list` (*`oid`*, *`arr`*[, *`dflt`*])

This function fills the associative array *arr* with a list of attributes set for the element identified by *oid*. This function returns the number of attributes in the array or `0` if the element has no attributes. Each attribute is stored in the array using the attribute name as the key, and its value as the array element value.

> **Note**
>
> Even if you have applied an alias map to the document, this function will return an attribute's real name, not its alias.

dflt is an optional third argument, which if non-zero, results in all attributes for the element identified by *oid* to be returned. For those attributes whose values are unspecified on the start tag *oid*, the default values are returned. For namespaced tags, `oid_attr_list` returns the namespace and any undefined attributes.

For example, the `oid_attr` function could be written using this function as:

```
function oid_attr(oid, attrname)
```

```
{
 local al[]
 oid_attr_list(oid, al)
 return (attrname in al) ? al[attrname] : ""
}
```

The built-in function is more efficient, however, since it does not need to create an array using `oid_attr_list` to obtain the value.

oid_attr_required

`oid_attr_required` (*`oid`*, *`attrname`*)

This function returns `1` (True) if the attribute *attrname* for the element specified by OID is a required attribute, that is, declared as #REQUIRED in the DTD.

Note

If you have applied an alias map to the document, *attrname* can be either an alias or a real name.

oid_attr_type

`oid_attr_type` (*`oid`*, *`attrname`*)

This function returns the declared attribute value type for the attribute named *attrname* of the element specified by *oid*.

Note

If you have applied an alias map to the document, *attrname* can be either an alias or a real name.

If the attribute was declared as a name group, this is a string of the form "CDATA", "ENTITY", "ENTITIES", "ID", "IDREF", "IDREFS", "NAME", "NAMES", "NMTOKEN", "NMTOKENS", "NOTATION", "NUMBER", "NUMBERS", "NUTOKEN", "NUTOKENS", or "NAMEGROUP".

This function is equivalent to:

```
tag_attr_type(oid_name(oid), attr, oid_doc(oid))
```

oid_backward

`oid_backward` (*`oid`*)

This function returns the oid of the element preceding the element specified by *oid* in linear (galley strip) order. Returns `oid_null` if *oid* is the first object in the document.

Examples

The `oid_backward` function could be written using `oid_prev` and `oid_child` as follows:

```
function oid_backward(o)
{
 if (!oid_valid(o)) {
 return oid_null()
}
# if no left sibling, return parent
local lsib = oid_prev(o)
if (!oid_valid(lsib)) {
 return oid_parent(o)
}
# return deepest, rightmost child of the
# immediate left sibling, if there is one
local rchild = lsib, last
while (oid_valid(rchild)) {
 last=rchild
 rchild=oid_child(rchild, -1)
}
return last
}
```

oid_caret

`oid_caret` (`[`*`doc`*`]`)

This function returns the object identifier of the object containing the cursor. For singleton tags, this function returns the object identifier of the singleton immediately to the left of the cursor. If the cursor is outside the outermost element or if the document has no objects (that is, an ASCII file or an empty document), this function returns an invalid OID (that is, `oid_null`). The `oid_caret` takes an optional argument, *doc*, that specifies the identifier of the document tree. If *doc* is omitted or is `0`, then the current document is used.

oid_caret_offset

`oid_caret_offset` (`[`*`oid`*`]`)

This function returns the position of the cursor in characters from the specified *oid*, or `oid_caret` if *oid* is not supplied. This function is the same as `oid_caret_pos` except that it always returns a character offset. Each non-text object (that is, markup, equations, tables, graphics) counts as one character. `oid_caret_pos` will return `-1` as a special case if the cursor is positioned before the

end tag of the object, since this is more efficient than calculating the offset. The offset is sometimes needed, however, if characters or other objects are inserted after the cursor.

oid_caret_pos

oid_caret_pos ([*oid*])

This function returns the position of the cursor in characters from the specified *oid*, or `oid_caret` if *oid* is not supplied. `oid_caret_pos` will return `-1` as a special case if the cursor is positioned before the end tag of the object, since this is more efficient than calculating the offset. Each non-text object (that is, markup, equations, tables, graphics) counts as one character. This position may be used with `oid_caret` and `goto_oid` o restore the cursor to a previous location in the document, for example:

```
oid = oid_caret()
pos = oid_caret_pos(oid)
 ...
goto_oid(oid, pos)
```

oid_check_attr

oid_check_attr (*oid*, *attr*, *value*)

This function returns a null string if *value* specifies a valid setting for the attribute named *attr* for the element identified by *oid*. If *value* is illegal, it returns a message describing the error.

Note

If you have applied an alias map to the document, *attr* and *value* can either be aliases or real names.

oid_child

oid_child (*oid*[, *n*])

This function returns the OID of the *n*th child of the element identified by *oid* or `oid_null` if there is no such child. If *n* is omitted, then the OID of first child is returned. If *n* is `-1`, then the OID of the last child is returned.

oid_children

oid_children (*oid*)

This function returns the number of children of the element specified by OID or `0` if the element contains no elements.

Examples

This function can be written using the `oid_child` primitives as follows:

```
function oid_children(parent)
{
 local n = 0;
 local o = oid_child(parent)
 while (oid_valid(o)) {
 n++
 o = oid_next(o)
}
 return n;
}
```

oid_content

`oid_content` (*`oid`*`[, `*`flags`*`]`)

This OID function returns the content of the element specified by *oid* as a markup-coded string.

The optional argument *flags* is a bitmask that controls the markup included in the returned string and is constructed using OR with the following flags:

- `0x1` — Include the start tag (and end tag if not EMPTY) markup for *oid* in the string. If not specified, then only the content is included.
- `0x2` — Include only the start tag for *oid* markup if applicable (no content nor end tag). If this bit is specified, then the `0x1` bit is ignored.
- `0x4` — Don't convert non-ASCII characters to entity references (as though `set writenonasciichar=char` option was in effect). If this bit is not specified, non-ASCII characters are converted based on the `set writenonasciichar` command on page 933 setting.
- `0x8` — Suppress insignificant line breaks, which are line breaks that occur where a record end is not significant. If this bit is not specified, insignificant line breaks are included based on the `set outputrecordlength` command on page 858 setting.
- `0x10` — Suppress Arbortext processing instructions. Arbortext processing instructions can also be suppressed using the `set writepi` on page 935 command.
- `0x20` — Use XML syntax in the string returned even if the selection is in an SGML document.
- `0x40` — Use SGML syntax in the string returned even if the selection is in an XML document.

- `0x80` — Do not include the markup in the result. This option takes precedence over any other flag bits that control how markup is generated.
- `0x100` — For XInclude elements, expand the element and return its content only (do not include the `<xi:include>` tag).

Because of memory requirements, you should not request the content of large elements such as the entire document, chapters, or sections.

Example:

Consider the following string of tagged content. In these examples, assume that your text caret is within this paragraph.

```
<para id="P1">This is a test.</para>
```

To return the content only, use the following function call:

```
oid_content(oid_caret())
```

To return the content and the surrounding tags, use the following function call:

```
oid_content(oid_caret(),1)
```

To return the start tag markup only, use the following function call:

```
oid_content(oid_caret(),2)
```

oid_content_model

oid_content_model (*oid*[, *doc*])

This function returns the canonical form of the content model for the element specified by *oid*.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

oid_context_string

oid_context_string (*oid*[, *include*])

This OID function returns the context string of the element specified by *oid*. If the optional parameter *include* is given and non-zero, the start tag (and end tag if not EMPTY) for *oid* is included in the string.

oid_current_tag

oid_current_tag ([*doc*[, *forward*]])

In the Edit window view, this function returns the identifier of the tag object that is backward from the cursor in the document if *forward* is `0` or omitted. If *forward* is non-zero, `oid_current_tag` returns the identifier of the tag object that is forward from the cursor.

Note

If *forward* is `0` or omitted, the object returned corresponds to the tag that would be affected by the `modify_tag` command on page 684.

In the Document Map view, this function returns a tag object identifier based on the cursor location and the value of the `set docmapcurrenttag` on page 782 command.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

Unlike `oid_caret` this only returns tag objects, never equation images. Also unlike `oid_caret`, if the encountered tag is an end tag, `oid_current_tag` returns the OID of its start tag instead of the parent object.

oid_declared_tag

`oid_declared_tag` (*`tagname`*`[,` *`oid`*`[,` *`doc`*`]])`

This function returns `1` (true) if *tagname* is the name of an element declared in the document type (defined in a DTD or schema). The function returns `0` (false) when:

- *tagname* is not a valid tag name, or
- *tagname* is a Arbortext Editor supplied tag (for example, the processing instructions `_font` or `_ignore`),or
- the tag is from a foreign namespace.

tagname is looked up with respect to the position in the document represented by *oid*. If *oid* is not supplied, the current caret point of the current document is assumed. If *oid* is supplied and is `oid_null()`, *tagname* is looked up independent of any document instance and is only looked up with respect to the document type of *doc*.

Note that the *doc* parameter is only used if *oid* is supplied and is `oid_null()`. If *doc* is not supplied or is `0`, the current document is used.

oid_delete

`oid_delete` (*`oid`*`[,` *`flags`*`])`

This function deletes the object identified by *oid* and its content depending on the *flags* parameter.

The optional *flags* parameter is a bitmask that controls what is deleted and is constructed using OR with the following flags:

- `0x1` — delete only the tag (and end tag if not EMPTY).
- `0x2` — delete only the content if an element.

If *flags* is omitted or 0 both the element and its content are deleted. If *oid* does not specify an element, only the object is deleted.

oid_delete_attr

`oid_delete_attr` (*`oid`*, *`attrname`*)

This function deletes the attribute named *attrname* from the element specified by *oid*.

> **Note**
>
> If you have applied an alias map to the document, *attrname* can be an alias, as well as a real name.

Returns `1` (True) if the attribute was deleted, or `0` (False) if *oid* is invalid or does not have an *attrname* attribute. If *attrname* specifies a required attribute, then the document is marked incomplete.

oid_detail

`oid_detail` (*`oid`*, *`flag`*)

This function can expand or collapse an element in the Edit window. The element is specified by *oid*.

If *flag* is `1`, the content of the element specified by *oid* is replaced by a plus sign (+) enclosed in a box in the Edit window (in other words, the element is collapsed). If flag is `0`, the element is expanded. Note that this function does nothing when the specified *oid* is a division heading element and the current window is the Edit window.

As shown below, the element containing the *oid* (the caret in this case) will be collapsed.

```
oid_detail(oid_caret(), 1)
```

Use `0` instead of `1` to expand detailing.

The `oid_expose` function can also be used if the Edit window is synchronized with the Document Map window.

oid_detailed

`oid_detailed` (*`oid`*)

This function returns `1` (True) if the contents of the element identified by *oid* is detailed. Detailed means it's represented by a plus sign (+) enclosed in a box in the Edit window (in other words, the element is collapsed).

oid_dialog

`oid_dialog` (*`oid`*)

This function returns the window ID of the XUI dialog associated with the specified *oid*. The dialog can be either the embedded ActiveX dialog associated with an intelligent graphic or a dialog that is associated with the element referenced by the *oid* through the document type configuration file (`.dcf` file). If the dialog does not exist, the function returns `-1`.

Refer to the *Customizer's Guide* for more information about XUI dialogs.

oid_doc

`oid_doc`(*`oid`*)

This function returns the document ID for the specified *oid*.

oid_effective_dita_default_attrs

`oid_effective_dita_default_attrs`(*`oid`*, *`array[]`*)

For DITA documents, this function populates an *array* with all of the effective default attributes at the specified *oid*. The effective attributes include document type specified default values, processor default values, and inherited values. This function does not return explicitly set attribute values. If the given *array* is not empty, only those attributes with names that match keys in the array will be populated. Otherwise, all attributes with default values are populated.

For example, assume you have the following markup in a DITA map:

```
<topicgroup format="ditamap">
 <topicref format="dita" href="myTopic.dita"/>
</topicgroup>
```

Calling this function for the `topicref` element would return `ditamap` for the *format* attribute. It would not return `dita`, because that is an explicitly set value instead of the default value.

oid_effective_dita_attr

oid_effective_dita_attr(*oid*, *attr*)

For DITA documents, this function returns the value of the specified attribute *attr* at the specified *oid*. The attribute values the function can return include explicitly set values, document type specified default values, processor default values, and inherited values. For example, assume you have the following markup in a DITA map:

```
<topicgroup format="ditamap">
 <topicref href="myMap.ditamap"/>
</topicgroup>
```

Calling this function for the *format* attribute on the `topicref` element would return `ditamap`.

oid_effective_dita_attrs

oid_effective_dita_attrs(*oid*, *array[]*)

For DITA documents, this function populates an *array* with all of the effective attributes at the specified *oid*. The effective attributes include explicitly set values, document type specified default values, processor default values, and inherited values. For example, assume you have the following markup in a DITA map:

```
<topicgroup format="ditamap">
 <topicref href="myMap.ditamap"/>
</topicgroup>
```

Calling this function for the `topicref` element would return `ditamap` for the *format* attribute.

oid_empty

oid_empty (*oid*)

This function returns 1 (True) if the element identified by *oid* has no content.

oid_encl_include

oid_encl_include (*oid*[, *pos*])

This function returns the *oid* of the of the nearest enclosing included object to the location specified by *oid* and *pos*, regardless of whether or not the object is expanded. If the specified *oid* is not in an included object, this function returns a null string.

pos can be any of the following values:

- `0` — (The default.) *pos* is immediately after the start tag identified by *oid*.
- `-1` — *pos* is immediately before the end tag of the element identified by *oid*.
- `-2` — *pos* is immediately before the start tag.
- `-3` — If *oid* is a start tag, *pos* is immediately after the end tag. Otherwise, *pos* is at the end of the object.

oid_entity_first

oid_entity_first (*oid*)

This function returns the OID of the first element within the text or file entity specified by OID *oid*. This function returns the null OID if *oid* is not a file or text entity, or if there are no elements within the entity. If *oid* is a file entity reference, the OID returned would be the top of a different document tree than the one *oid* is in. If *oid* is a text entity, the OID returned is in the same doc as the one *oid* is in.

oid_entity_last

oid_entity_last (*oid*)

This function returns the OID of the last element within the text or file entity specified by OID *oid*. This function returns the null OID if *oid* is not a file or text entity or if there are no elements within the entity. If *oid* is a file entity reference, the OID returned is in a different document tree than the one *oid* is in. If *oid* is a text entity, the OID returned would be in the same document as the one *oid* is in.

oid_entity_lock

oid_entity_lock (*oid*[, *resultcode*[, *showerror*[, *pos*]]])

This function locks a file entity or XML inclusion for editing and calls entitylockhook on page 950 or includelockhook on page 958 if appropriate. The function only operates if the given *oid* at cursor position *pos* is in a file entity or XML inclusion, and if the entity or inclusion is not stored in a Document Management System (DMS). DMS objects must be checked out according to the user's system administration practices before they can be edited.

If the *oid* is in an entity or inclusion (or is an entity or inclusion referencing itself) and the entity or inclusion is not stored in a DMS, then the entity or inclusion will be locked for editing. If an `entitylockhook` is defined it will be called first and the entity or inclusion will be locked only if the hook returns a value of `0`. `oid_entity_lock` returns `0` only if it tries to lock an entity or inclusion and fails. In all other cases this function will return `1`.

resultcode, if supplied, must be a variable set to a code indicating the status of the lock operation. The possible values are:

- `0` — Locked and not refreshed.
- `1` — The entity or inclusion was locked, but had been modified on disk since it was opened and was reloaded.
- `2` — The entity or inclusion could not be locked because someone else has it locked.
- `3` — The entity or inclusion could not be locked because the file it is in is read only.
- `4` — The entity or inclusion could not be locked for some other reason.

If *showError* is supplied and non-zero, an error message will be displayed for any failure to lock the include or entity.

pos can be any of the following values:

- `0` — (The default.) *pos* is immediately after the start tag identified by *oid*.
- `-1` — *pos* is immediately before the end tag of the element identified by *oid*.
- `-2` — *pos* is immediately before the start tag.
- `-3` — If *oid* is a start tag, *pos* is immediately after the end tag. Otherwise, *pos* is at the end of the object.

oid_expose

`oid_expose` (*`oid`*`[,` *`newval`*`[,` *`descend`*`]])`

This function returns `1` (True) if the content for the element identified by *oid* are displayed in the Document Map window. Otherwise, it returns `0`.

If *newval* is specified, it changes the state of the content display for *oid*. If `0`, the content is collapsed. If non-zero, the content lines for the element are displayed.

If *descend* is specified and non-zero, then `oid_expose` changes the exposed state of all children of *oid*, recursively.

Note, the effect of this function is visible in the Document Map and Column view only. The expose bit is independent of detailing, which is shown only in non-Document Map or Column views.

oid_find_child_attrs

`oid_find_child_attrs` (*`oid`*`,` *`arr`*`,` *`attr`*`[,` *`value`*`[,` *`flags`*`]])`

This function fills the array *arr* with all the oids of the descendents of *oid* that contain the attribute *attr* with the value of *value*. If OID is *oid_null()*, then the entire document tree for the current document is searched. If *value* is null, then the

descendent is returned if *attr* is set to any value. The *flags* parameter is a bitmask specifying the search options, and is constructed by using OR with the flags from the following list:

- `0x01` — Return after first match
- `0x04` — Match case (search case-sensitive)
- `0x08` — The supplied *value* is a regular expression (flag `0x04` is ignored)
- `0x10` — Match only attributes of declared type `ID`. If *attr* and *value* are both null, then all elements having an `ID` attribute are returned.
- `0x20` — Match only attributes of declared type `IDREF` or `IDREFS`. If *attr* and *value* are both null, then all elements having an `IDREF` or `IDREFS` attribute are returned. This bitmask is ignored if the document is freeform XML.

This function returns the number of oids stored in the array `arr`.

oid_find_children

`oid_find_children` (*`oid`*, *`arr`*, *`tag`*[, *`flags`*])

This function fills the array *arr* with all the oids of the descendents of *oid* that match the tag name *tag*. If OID is *oid_null( )*, the entire document tree for the current document is searched. The *flags* parameter is a bitmask specifying the search options, and is constructed by ORing the flags from the following list:

- `0x01` — Return after first match
- `0x04` — Match case (search case-sensitive)
- `0x08` — The supplied *tag* is a regular expression (flag `0x04` is ignored)

This function returns the number of oids stored in the array `arr`.

oid_find_parent_attrs

`oid_find_parent_attrs`(*`oid`*, *`arr`*, *`attr`*[, *`value`*[, *`flags`*]])

This function fill the array *arr* with all the oids of the ancestors of *oid* that contain the attribute *attr* with the value of *value*. If *value* is null, then the ancestor is returned if *attr* is set to any value. The *flags* parameter is a bitmask specifying the search options, and is constructed by using OR for the flags from the following list:

- `0x01` — Return after first match
- `0x02` — Ascend into ancestor file entities (this flag will not function properly if the file entity is open in its own window)
- `0x04` — Match case (search case-sensitive)

- `0x08` — The supplied *value* is a regular expression (flag `0x04` is ignored)
- `0x10` — Match only attributes of declared type `ID`. If *attr* and *value* are both null, then all elements having an `ID` attribute are returned. This bitmask is ignored if the document is freeform XML.
- `0x20` — Match only attributes of declared type `IDREF` or `IDREFS`. If *attr* and *value* are both null, then all elements having an `IDREF` or `IDREFS` attribute are returned. This bitmask is ignored if the document is freeform XML.

This function returns the number of oids stored in the array `arr`.

oid_find_valid_insert

`oid_find_valid_insert``(`*`oid`*`[, `*`offset`*`[, `*`flags`*`]])`

This function returns the nearest valid insertion point if the `oid` passed in is within generated text, a protected region, a text entity, or a protected change tracking region. The *offset* parameter is modified indirectly. The optional *flags* parameter controls the search and defaults to `0` (search forward).

- `0x1` — *backward* (search backward if set, otherwise forward)
- `0x2` — *autoreverse* (if search fails try the other direction)
- `0x4` — Returns an insertion point that is not read-only even if the location does not allow markup such as a comment or CDATA section. Otherwise, the function will disallow these areas.

If no valid insertion point is found, this function returns the null `oid`.

oid_first

`oid_first` `([`*`doc`*`])`

This function returns the OID of the first object in the document. `oid_first` takes an optional argument, *doc*, that specifies the identifier of the document tree. If *doc* is omitted or `0`, then the current document is used.

oid_first_tag

`oid_first_tag` `([`*`doc`*`])`

This function returns the OID of the first element start tag in the document. `oid_first_tag` takes an optional argument, *doc*, that specifies the identifier of the document tree. If *doc* is omitted or `0`, then the current document is used.

Use `oid_first_tag` instead of `oid_first` on page 441, when you do not want comments and processing instructions at the beginning of the document.

oid_forward

oid_forward (*oid*)

This function returns the object identifier of the element following the element specified by *oid* in linear (galley strip) order. This function returns `oid_null` f *oid* is the last element in the document.

Examples

The loop shown in the following excerpt from a function lists all elements in the document instance from first to last:

```
for (o = oid_first(); oid_valid(o); o = oid_forward(o)) {
 eval oid_name(o) output=>*
}
```

The `oid_forward` function can be written using `oid_next` and `oid_child` as follows:

```
function oid_forward(o1)
{
 if (!oid_valid(o1)) { return oid_null();}
 local o = oid_child(o1);
 if (oid_valid(o)) { return o;}

 while (oid_valid(o1)) {
 o = oid_next(o1);
 if (oid_valid(o)) { return o;}
 o1 = oid_parent(o1)
}
 return oid_null()
}
```

oid_gentext

oid_gentext (*oid*[, *after*])

The `oid_gentext` function returns the generated text for the *oid* as an SGML string. It returns the null string if the oid does not exist or does not have any generated text.

The *after* parameter specifies that generated text that follows the element's content (that is with `placemnt=after`) should be returned. If this parameter is omitted (or `0`), generated text that precedes the element content, (that is with `placemnt=before`) is returned.

oid_gentext_source

oid_gentext_source (*oid*)

The `oid_gentext_source` function returns the object identifier of the source element that was used in `#CONTENT` to create the generated text object specified by *oid*. If *oid* is not a generated text OID or is not from `#CONTENT`, `oid_gentext_source` returns `oid_null()`.

The `oid_gentext_source` function is intended to be called from functions that are called as a result of the `attloc=system-func` FOSI setting.

oid_get_icon

`oid_get_icon` (*`oid`*[, *`index`*])

This function returns the name of the display icon that appears to the left of the specified *oid*. The icons appear in the Document Map and also appear in the Edit window. However, whether the icons appear in the Edit window is dependent on the settings of the `set showiconsfulltags` on page 887, `set showiconspartialtags` on page 888, and `set showiconsnotags` on page 887 commands. A list of returnable icon names is available in the help topic for the `oid_set_icon` on page 456 function.

An element can have one or two icons associated with it through `oid_set_icon`. The *index* parameter may be set to `1` or `2` to retrieve the first or second icon, respectively. If *index* is omitted, then the first icon is returned.

If the specified *oid* does not have the specified icon, *index* is something other than `1` or `2`, or *oid* is invalid, the function returns the string `None`.

The following example gets the icon for the oid to the left of the cursor:

```
$icontype = oid_get_icon(oid_current_tag())
```

oid_graphic_current_view

`oid_graphic_current_view`(*`oid`*)

This function returns the current view displayed for the EDZ or ISO intelligent graphic referenced by *oid*. A view is a custom display of the intelligent graphic that was created and saved inside of the graphic. If *oid* is invalid or does not refer to an intelligent graphic, the function returns an empty string. If the current view of the intelligent graphic is the default view, the function also returns an empty string.

oid_graphic_format

`oid_graphic_format` (*`oid`*)

This function returns the graphic file format of the graphic referenced by *oid*. If the *oid* is invalid or does not reference a graphic, the function returns an empty string.

The returned value can be one of the following formats:

- `sunbm` — Sun Bitmap
- `xbm` — X Bitmap
- `eps` — Encapsulated PostScript file
- `cur`— Windows Cursor file
- `drw` — IslandDraw DRW file
- `sunicon` — Sun Icon file
- `tif` — Tag Image File Format
- `drwunc` — IslandDraw DRAW file (not compressed)
- `png` — Portable Network Graphics
- `wmf` — Windows metafile
- `icon` — Windows icon
- `pcx` — PC Paintbrush File Format
- `bmp` — Windows Bitmap
- `cgm` — Computer Graphics Metafile
- `gif` — Graphic Interchange Format
- `jpg` — Joint Photographic Exports Group
- `calsG4` — CALS raster (G4)
- `svg` — Scalable Vector Graphics
- `idr` — Arbortext IsoDraw file
- `idrz` — Arbortext IsoDraw file
- `iso` — Arbortext IsoDraw file
- `isoz` — Arbortext IsoDraw file
- `edz` — Creo View file
- `pvz` — Creo View file

oid_graphic_pathname

`oid_graphic_pathname` (*`oid`*)

If *oid* is a graphic tag, this function returns the path name for the graphic. If *oid* is invalid, or is not the *oid* for a graphic tag, this function returns the null string. If the path name comes from a tag attribute value, this function expands character and entity references.

The function attempts to resolve relative paths using the base URI for the location where the graphics element is located before attempting to resolve the path using the document directory or directories from the `graphicspath` on page 828. The base URI and the document directory are often the same, but may be different when the graphic element is contained in an XInclude, file entity, or other included content.

In addition, if the graphics reference value is a relative filename that cannot be resolved to an absolute path name because the file does not exist, the string returned will be the absolute path name for the document directory concatenated to the relative name. This prevents the relative name from being inadvertently resolved to a file in the current working directory. However, if the document in question has not been saved, there is no document directory and the graphic name (which may be an absolute or a relative name) will be returned as is.

Consider the following markup:

```
<inlinegraphic
 fileref="http://server/cgi-bin/script?param1=value1&amp;param2=value3"/>
```

If the *oid* for this tag is contained in the variable `o`, and you call `oid_graphic_pathname`:

```
local path = oid_graphic_pathname( o )
```

then the path name would contain the string:

```
http://server/cgi-bin/script?param1=value1&param2=value3
```

Note that the `&` character entity reference was expanded to the `&` character.

oid_graphic_size

`oid_graphic_size` (*`oid`*, *`width`*, *`height`*, *`switch`*)

This function returns the size of the image displayed for the graphic or equation identified by *oid*. The image width in pixels is returned in the output variable given by *width*, the height in pixels in the output variable given by *height*.

If *switch* is `0` (the default), `oid_graphic_size` returns the displayed size of the graphic, which may be scaled. If *switch* is `1`, then `oid_graphic_size` returns the defined size of the graphic.

This function returns `1` (true) if it succeeds, and `0` (false) if it fails, for example, if *oid* is not an equation or graphic. If it fails, the function writes an error message to the $ERROR predefined variable on page 78.

oid_graphic_viewer

`oid_graphic_viewer` (*`oid`*)

This function returns the name of the embedded ActiveX dialog used to display the intelligent graphic identified by *oid*. If the intelligent graphic is an EDZ graphic and displayed by ProductView, the function returns `pview`. If the

intelligent graphic is an ISO graphic and displayed by Arbortext IsoView, the function returns `iview`. If *oid* does not reference an intelligent graphic, the function returns an empty string.

oid_has_attr

`oid_has_attr` (*`oid`*, *`attrname`*)

This function returns `1` (True) if the start tag identified by *oid* specifies a value for the attribute named *attrname*. If the start tag does not specify a value for *attrname*, or if *oidattrname* is not a valid attribute name, this function returns `0`.

Note

If you have applied an alias map to the document, *attrname* can be either an alias or a real name.

This is different from the function `tag_has_attr` that tests if a given attribute is declared for the element in the DTD.

oid_id_attr_name

`oid_id_attr_name` (*`oid`*)

This function returns the name of the ID attribute for the element specified by *oid*. `oid_id_attr_name` returns the null string if the *oid* is not valid or the element does not declare an ID-type attribute.

Note

Even if you have applied an alias map to the document, this function will return an attribute's real name, not its alias.

oid_in_doc

`oid_in_doc` (*`oid`*, *`doc`*)

This function returns `1` (True) if *oid* belongs to the document tree specified by *doc*.

oid_include_expand

oid_include_expand (*oid*[, *pos*[, *flags*]])

This function returns the state of the nearest enclosing included object for the specified *oid* and character location *pos*, either expanded or collapsed.

If *flags* is omitted, and the specified *oid* is not in an included object, this function returns a `-1`. If *flags* is omitted, this function returns a `1` if the enclosing object is expanded, and a `0` if it is collapsed.

The *flags* parameter is a bitmask that controls the following options. It is constructed by using OR for the flags from the following list:

- `0x01` — Causes the include to be expanded (if it is not already) to show the contents of the included object. If not set, the include will be collapsed (if it is not already) to show the xinclude markup itself
- `0x02` — Prompts the user to save any unsaved changes if the collapse or expand would cause those changes to be lost. If not set, no prompting or saving occurs.

pos can be any of the following values:

- `0` — (The default.) *pos* is immediately after the start tag identified by *oid*.
- `-1` — *pos* is immediately before the end tag of the element identified by *oid*.
- `-2` — *pos* is immediately before the start tag.
- `-3` — If *oid* is a start tag, *pos* is immediately after the end tag. Otherwise, *pos* is at the end of the object.

oid_invalid_markup

oid_invalid_markup (*oid*)

This function returns `0` if the start tag identified by *oid* contains no invalid markup. This means it is a foreign namespace element, or it is a declared element and all of its attributes are declared and have valid values.

If this function does not return `0` then the result is a bit mask that can be inspected to determine which parts of the tag are invalid.

- 0x01— This bit will be set if the element is not declared and is not a foreign namespace element.
- 0x02— This bit will be set if a declared attribute has a value which is invalid for that attribute type.
- 0x04— This bit will be set if there are attributes which are undeclared for that element.

Example:

```
function give_invalid_status(oid)
{
 local oim
 local result
 oim = oid_invalid_markup(oid)
 result = ""
 if (oim == 0){
 result .= "Start tag contains no invalid markup."
} else {
 if (oim & 0x01) {
 result .= "Start tag is undeclared. "
}
 if (oim & 0x02) {
 result .= "Start tag contains an invalid \
 attribute value for a declared attribute. "
}
 if (oim & 0x04) {
 result .= "Start tag contains an undeclared \
 attribute. "
}
}
 response(result)
}
```

oid_invalidate_graphic

oid_invalidate_graphic(*oid*)

This function resolves, reloads, and redraws (if displayed) the graphic associated with *oid*. This function does not modify the document, so it can be used with a read-only document or an unlocked entity.

oid_is_gentext

oid_is_gentext (*oid*)

This function returns a one (1) if *oid* is valid and is generated text. This function returns a zero (0) if *oid* is invalid, or is part of the loaded document instance.

oid_last

oid_last ([*doc*])

This function returns the object identifier of the last object in the document tree given by *doc*, or the current document if *doc* is omitted or `0`.

Examples

This function could be written using `oid_child` as follows:

```
function oid_last(doc = 0)
{
 local o = last = oid_first(doc)
 # just go right as deep as we can
 while (oid_valid(o)) {
 last = o
 o = oid_child(o, -1); # get last child
}
 return last
}
```

However, using the built-in `oid_last` function is more efficient.

oid_level

oid_level (*oid1*, *oid2*)

This function returns the number of levels of ancestry between the elements specified by *oid1* and *oid2* or `0` if the elements are not related, that is, neither OID contains the other. If *oid1* contains *oid2*, then the level number is positive. If *oid2* contains *oid1*, then the level number is negative.

Examples

This function could be written using `oid_parent` as follows:

```
function oid_level(o1, o2)
{
 local i, p, o
 if (o1 == o2) { return 0;}
 if (o1 < o2) {
 o = o1
 p = oid_parent(o2)
} else {
 o = o2
 p = oid_parent(o1)
}
 # walk up parent tree until o is found
 for (i = 0; oid_valid(p); ) {
 i++;
 if (p == o)
 { return o == o1 ? i : -i;}
 p = oid_parent(p)
}
 return 0; # not related
}
```

oid_logical_mate

oid_logical_mate (*oid*)

This function returns the logical mate of an oid. The logical mate is a singleton that acts as a mate to the first parameter, but is not an end tag in the XML structure. The function can also return the start mate that matches an end mate.

The function returns the null oid if the argument is not a valid oid, an oid that cannot take a logical mate, or if the logical mate is not present in the document. The latter condition means that the document is semantically incomplete or unbalanced.

oid_modified

oid_modified (*oid*[, *value*])

This function returns the current "modified" state for the specified *oid*. If the specified *oid* has been modified since the last save operation (or since `oid_modified` was used to set the oid to "unmodified"), the function returns a one (1). If the oid has not been modified, the function returns a zero (0).

If the optional *value* parameter is included, the function still returns the current "modified" state of the specified *oid*, but also sets the "modified" state to the specified *value*. The *value* parameter must be either a one (1) for "modified", or a zero (0) for "unmodified."

Examples:

```
$mod = oid_modified(oid_current_tag())
$ret = oid_modified(oid_next(),1)
```

oid_modify_attr

oid_modify_attr (*oid*, *[attrname]*, *[value]*)

This function sets the attribute named *attrname* to *value* for the element specified by *oid*. If you do not specify either *attrname* or *value*, the **Modify Attributes** dialog is raised.

> **Note**
>
> If you have applied an alias map to the document, *attrname* and *value* can be either aliases or a real names.

Returns `1` (True) unless:

- there is no such attribute for the element, or
- *oid* is invalid, or
- the document is read-only

`oid_modify_attr` can be used to assign XML namespace attributes to elements in an XML document.

oid_mouse_pos

`$oid = oid_mouse_pos (pos[, doc[, flag]])`

This function returns the oid corresponding to the current mouse position. If the mouse pointer is pointing directly at the begin or end tag of an element, the oid returned corresponds to that element.

The *pos* parameter is an output variable to hold the character offset from the element's start tag to the character closest to the mouse pointer.

The *doc* parameter specifies the document to operate upon. If *doc* is omitted or `0`, the value returned by `current_doc` on page 255 is used.

When the mouse pointer is not pointed directly at the begin or end tag of an element, the value of *flag* has no effect. But when the mouse pointer is pointed directly at the begin or end tag of an element, *flag* works as follows:

- If *flag* is `1`, then `oid_mouse_pos` evaluates whether the mouse pointer is closer to the left or right side of a tag. The function will return the oid that is outside the tag pair if the mouse is pointing at the left half of the begin tag or the right half of the end tag.
- If *flag* is omitted or `0`, then `oid_mouse_pos` returns the oid which is inside the tag under the mouse pointer.

The following code will produce a result similar to the command `caret mouse`, unless the mouse is pointing directly at a begin or end tag. If the mouse is pointing directly at a begin or end tag, the caret will always be positioned inside the tag pair. With the `caret mouse` command, however, the caret will be positioned outside the tag pair if the mouse is pointing at the left half of the begin tag or the right half of the end tag.

```
local oid, pos
oid = oid_mouse_pos(pos)
goto_oid(oid, pos)
```

oid_name

`oid_name` (*`oid`*)

This function returns the name of the element identified by *oid* or the null string if the *oid* is not valid. If this function returns `_file_ent`, you can use the `entity_name` on page 344 function to find the name of the entity.

> **Note**
>
> Even if you have applied an alias map to the document, this function will return an element's real name, not its alias.

oid_namespace_prefix

`oid_namespace_prefix` (*`oid`*, *`uri`*)

This function returns the prefix for the *uri* that is active at the specified *oid*. If no prefix is declared for the namespace URI, then the function returns a null string.

oid_namespace_prefix_defined

`oid_namespace_prefix_defined`(*`oid`*, *`uri`*)

This function determines whether a *uri* is defined at an *oid*. If the given *uri* is not defined at the specified *oid*, then the function returns a `0`. If the given *uri* is defined at the specified *oid*, then the function returns a `1`.

Following is some example code using this function:

```
if (oid_namespace_prefix_defined(oid, uri)) {
 local prefix = oid_namespace_prefix(oid, url);
 if (prefix == "") {
 # This namespace is the default namespace at this oid.
}
 else {
 # This namespace is not the default namespace at this oid.
 # However, it is defined and has a prefix whose value is in
 # the variable 'prefix'.
}
}
else {
 # This namespace is not defined at this oid.
}
```

oid_namespace_stack

`oid_namespace_stack` (*`oid`*, *`prefixArray`*, *`uriArray`*)

This function fills *prefixArray* and *uriArray* with the prefixes and URIs that are active at *oid*. This function returns the number of prefix and URI pairs stored in the arrays.

oid_namespace_uri

oid_namespace_uri (*oid*[, *prefix*])

This function returns the URI for the *prefix* that is active at *oid*. If *prefix* is not supplied, the function returns the URI of the active default namespace at *oid*.

oid_next

oid_next (*oid*)

This function returns the OID of the sibling following the element specified by *oid* or `oid_null` if *oid* is the last child.

oid_null

oid_null ()

This function returns an OID that represents no object. Several OID functions return `oid_null`; for example, `oid_parent` returns `oid_null` for the outermost element in a document. By definition, `oid_valid(oid_null)` is `0`.

oid_offset

oid_offset (*oid*, *pos1*, *oid2*, *pos2*)

This function returns the character offset between the two document locations specified by *oid1,pos1* and *oid2,pos2* respectively. Each non-text object (that is, markup, equations, graphics) counts as one character. The offset returned will be negative if the location specified by *oid2,pos2* comes before the position specified by *oid1,pos1*.

oid_parent

oid_parent (*oid*)

This function returns the OID of the immediate ancestor of the object specified by *oid* or `oid_null` if *oid* has no parent.

oid_paste_valid

oid_paste_valid (*oid*[, *pos*[, *pastedoc*]])

This function determines if it is valid to paste the contents of the paste buffer at the document location specified by *oid*, *pos*. If *pos* is omitted or `0`, the location to the right of the start tag identified by *oid* is used. The *pastedoc* argument specifies the document identifier of the paste buffer to test. If `0` or omitted, the default paste buffer is used.

`oid_paste_valid` returns one of the following integers:

- `2` — Paste is valid after fix ups, for example, adding required tags.
- `1` — Paste is valid.
- `0` — Paste is invalid. For example, the destination is protected, the region to paste is unbalanced, or context errors would result.
- `-1` — Paste may be valid, but involves pasting tags from a different document type, or the paste buffer is the default paste buffer and the system currently owns the clipboard.

Note, `oid_paste_valid` does not check if it is valid to paste tags from a different document type, for example, where the structures are same.

oid_prev

oid_prev (*oid*)

This function returns the OID of the sibling preceding the element specified by *oid* or `oid_null` if *oid* is the first child.

oid_prompt_attrs

oid_prompt_attrs (*oid*)

This function opens the **Modify Attributes** dialog box for the tag associated with *oid* if the tag has unspecified required attributes and the `requireattrs set` option is `on`, or if the tag has any declared attributes and the `promptattrs set` option is `on`.

If *oid* is not associated with a valid tag, the function returns `0`. If the dialog box is not required to be opened, or if the dialog box is opened and is accepted, the function returns `1`. If the dialog box is opened and the user cancels, the function returns `0` and the current operation is undone.

This function is useful when a tag needs to be inserted with prompts temporarily turned off, but then later the prompts are desirable.

oid_protected

oid_protected (*oid*)

This function returns 1 (True) if the content of the element specified by *oid* is protected so that no edits are allowed.

oid_read_only

oid_read_only (*oid*[, *value*])

This function returns the current "read only" state for the specified *oid*. The oid-level read only state is a feature that is not manipulated by the default Arbortext Editor features and functions. The oid-level read only state is only changed/used if an ACL programmer specifically uses this function to change the read only state.

If the specified *oid* has been set to a "read only" state, the function returns a one (1). If the oid has not been set to a "read only" state), the function returns a zero (0).

If the optional *value* parameter is included, the function still returns the current "read only" state of the specified *oid*, but also sets the "read only" state to the specified *value*. The *value* parameter must be either a one (1) for "read only", or a zero (0) for "not read only".

Examples:

```
$mod = oid_read_only(oid_current_tag())
$ret = oid_read_only(oid_next(),1)
```

oid_root

oid_root ([*doc*])

This function returns the OID of the root element (the first element start tag in a well formed document) in the document. oid_root takes an optional argument, *doc*, that specifies the identifier of the document tree. If *doc* is omitted or 0, then the current document is used.

oid_root returns oid_null on page 453 if the document is not well formed. (For example, if *doc* is a fragment that has multiple top level elements.)

oid_same_doc

oid_same_doc (*oid1*, *oid2*)

This function returns 1 (True) if *oid1* and *oid2* belong to the same document tree.

oid_select

oid_select (*oid*[, *invert*[, *primary*[, *include*]]])

This function marks the contents of the element specified by *oid*. The contents may be obtained by referencing the main::selection variable. This function returns `1` (True) if *oid* was marked or `0` (False) if the element has no content. If *invert* is `0`, then the marked region is not highlighted. The default is `1`, that is, the region is inverted.

If *primary* is `0`, then the primary selection is not claimed. The default is `1`, that is, the selection is asserted.

The optional fourth argument *include*, if non-zero, causes the start tag (and end tag if not EMPTY) for the element given by *oid* to be included in the selection. The default is `0`, that is, the element tags are not included.

This function returns the content of an element without affecting the screen display:

```
function oid_content(o)
{
 if (oid_empty(o)) { return "";}
 oid_select(o, 0, 0)
 return main::selection
}
```

oid_set_graphic_pathname

oid_set_graphic_pathname(*oid*, *path*)

If *oid* is a graphic tag, this function sets the path name for the graphic. If the graphic element supports a file reference attribute, that is used to set the path. If the element only supports an entity reference attribute, then a new graphic entity is declared and assigned to the attribute. When the parent document is a file system object, the new entity name is formed from the base name of the file (modified to be a valid entity name, if necessary). When the parent document is an object stored in a content management system, the new entity name is formed by prepending `&E` to a Universal Unique Identifier (UUID). If the *path* name is already declared as a graphic entity, then the existing entity name is used.

The *path* name value is stored as a relative path (as determined by `graphic_relative_path`) if possible, so that the reference is still valid if the document and graphic are moved to a different file system.

If *oid* is invalid, or is not the *oid* for a graphic tag, this function generates an error.

oid_set_icon

oid_set_icon (*oid*, *icon*[, *index*])

This function sets the display icon that appears to the left of the specified *oid*. The icons appear in the Document Map and also appear in the Edit window. However, whether the icons appear in the Edit window is dependent on the settings of the `set showiconsfulltags` on page 887, `set showiconspartialtags` on page 888, and `set showiconsnotags` on page 887 commands.

Specify the *icon* by specifying one of the supported icon names or by specifying a path and file name to a `.bmp`. If an invalid display icon is set, the built-in icon `None` will be displayed.

Specify a `.bmp` file using either a full or relative path. If you use a relative path, the `graphicspath` is searched. You can save icons in `.bmp` files in the *`Arbortext-path`*`\custom\graphics` directory. Icon graphics should have the BMP transparency set to pink. If the `.bmp` file is not found, then the behavior is as though no file was specified. The number of unique icons specified using a `.bmp` file are limited to 200. If this limit is reached, additional specified `.bmp` files behave as though no file was specified. If the `.bmp` is not found, the built-in icon `None` will be displayed.

You can use the *icon* names listed in the following table, which are case sensitive.

Available icon names

Name	Icon	Default use
`Attribute`		Indicates that an element has attributes assigned to it.
`BadAttribute`		Indicates that an element has attributes assigned to it and at least one of those attributes is not legal for the current document type definition.
`BadElement`		Indicates an element whose element name is not legal for the current document type definition.
`CheckedOut`		Indicates a document object accessed by a Repository Adapter is locked or checked out by the current user.
`Comment`		Indicates a comment element.
`Contracted`		Indicates that the element's contents are collapsed and hidden from view.
`DataMarkedSection`		Indicates that an SGML element is part of a data Marked Section
`DocObject`		Indicates a document object accessed by a Repository Adapter that is neither locked nor checked out.

Available icon names (continued)

Name	Icon	Default use
`Document`		Indicates a document.
`Element`		Indicates an SGML or XML element.
`Empty`		Indicates an element with no content.
`End`		Indicates an end tag.
`Equation`		Indicates an equation element.
`Expanded`		Indicates that the element's contents are expanded for viewing.
`FileEntity`		Indicates a referenced file entity.
`Graphic`		Indicates a graphic element.
`GrayCheckedOut`		Indicates a document object accessed from a Repository Adapter that is locked or checked out by the current user but is unavailable.
`GrayDocObject`		Indicates a document object accessed from a Repository Adapter that is neither locked nor checked out by the current user and is unavailable.
`GrayLocked`		Indicates a document object accessed from a Repository Adapter that is locked or checked out by another user and is unavailable.
`IgnoreMarkedSection`		Indicates that an SGML element is part of an ignored Marked Section
`Locked`		Indicates a document object accessed by a Repository Adapter is locked or checked out by another user.
`Missing`		Indicates where a required element is missing.
`None`		No icon displayed.
`ReadOnly`		Indicates a element that is not editable.
`Table`		Indicates a table element.

An element can have one or two icons associated with it. The *index* parameter may be set to `1` or `2` to set the first or second icon, respectively. If *index* is omitted, then the first icon is set.

If the specified *oid* is invalid or *index* is something other than `1` or `2`, the function returns a zero (0).

Otherwise, this function returns a one (1). If you specify an invalid *icontype*, the element is given an icon of `None` and the function still returns a one (1).

The following example sets the icon for the oid to the left of the cursor to the `Locked` icon.

```
$ret = oid_set_icon(oid_current_tag(),"Locked")
```

oid_show_attr

`oid_show_attr` (*`oid`*`[,` *`newval`*`]`)

This function returns 1 (True) if the attribute lines for the element identified by *oid* are displayed in the Document Map window. It returns 0, otherwise.

If *newval* is specified, it changes the state of the attribute display for *oid*. If `0`, the attribute lines are removed (collapsed). If non-zero, the attribute lines are displayed (if the element has any non-default attributes).

Note, the effect of this function is visible in the Document Map and Column view only.

oid_split_tag

`newoid =` **`oid_split_tag`**`(`*`[flags`*`[,` *`oid`*`[,` *`pos`*`]]])`

This function divides the element based on the location specified by *oid* and *pos*.

- *flags* — Optional. Sets options for the split operation. A value of one (`1`) specifies that all non-default attributes on the original element be duplicated on the newly created element. ID attributes (which should be unique across a document) are not duplicated. A value of zero (`0`) ensures that no attributes are duplicated on the newly created element. If the *flags* parameter is not specified, a value of `0` is assumed.
- *oid* — Optional. The oid of the element to split. If *oid* is omitted, the function uses `oid_caret`.
- *pos* — Optional. The character position at which to split. If *pos* is omitted, a value of `0` is assumed.

The `split` command is equivalent to calling the `oid_split_tag` function with no parameters. The `EditSplit` alias (used by the UI) is equivalent to calling `oid_split_tag` with a flags parameter of `1`.

oid_tbl_obj_list

`count=`**`oid_tbl_obj_list`**`(`*`oid`*`,` *`array`*`)`

This function fills *array* with the IDs of the table objects associated with *oid*, which must refer to a markup tag. It returns the number of IDs . Note that a zero return is not necessarily an error; not all OIDs have associated table objects.

The *array* is an associative array with the following indices:

set

grid

column

row

cell

rule

oid_top_pos

oid_top_pos (*pos*[, *doc*])

This function returns the OID of the object displayed in column 1 on the top line of the window containing the document specified by *doc*. If *doc* is omitted or `0`, the current document is used. The `oid_top_pos` function also sets the output variable *pos* to the character position following the top left column of the window.

This function is useful for restoring a screen position using `scroll_to_oid`, for example,

```
o = oid_top_pos(pos)
...
scroll_to_oid(o, pos)
```

oid_treeloc

oid_treeloc (*oid*[, *sep_char*[, *top_level_oid*]])

This function returns a string indicating the tree location in the current document of the tag specified by *oid*. For example, the returned string `1.3.2` refers to the second tag (`2`) of the third tag (`3`) of the first tag (`1`) in the document.

Specify a separator character *sep_char* to delineate the values in the string with a character other than ".", which is the default.

To limit the size of the returned value, specify a *top_level_oid*. This sets the parent tag at which the tree location will begin. If you do not specify a *top_level_oid*, the first value in the returned tree location represents the first tag in the document containing *oid*.

Note

If you specify an *oid* within a file entity, this function will not search back into the parent document .

oid_type

oid_type (*oid*)

This function returns a number that identifies the type of the object specified by *oid*. The return value is one of the following integers:

- `-1` — OID is invalid
- `0` — object is an element
- `1` — object is a user-defined tag alias of a DTD tag
- `4` — object is an equation image
- `7` — object is a processing instruction (includes Arbortext Editor-supplied tags)
- `8` — object is a user-defined alias of a processing instruction or unknown element
- `9` — object is an SGML comment
- `10` — object is an IGNORE marked section
- `11` — object is a CDATA marked section
- `12` — object is an RCDATA marked section
- `30` — object is a change tracking markup tag

oid_unknown

oid_unknown (*oid*)

This function returns `1` (True) if the start tag identified by *oid* is not a declared element in the DTD, or if the start tag is a declared element, this function returns `0`.

oid_unknown_attr_list

oid_unknown_attr_list(*oid*, *arr*)

This function fills the associative array *arr* with a list of unknown attributes set for the element identified by *oid*. This function returns the number of unknown attributes in the array or `0` if the element has no unknown attributes. Each unknown attribute is stored in the array using the attribute name as the key, and its value as the array element value.

Example:

```
function show_unknown_attrs(oid)
{
 local ual[]
 local count
 count = oid_unknown_attr_list(oid, ual)
 if (count == 0){
 response("The given oid has no unknown attributes.")
} else {
 local ua
 local i = 1
 local ual2[]
 # Convert from an associative array to a normal array
 # so it can be used with list_response().
 for (ua in ual) {
 ual2[i++] = ua . "=" . ual[ua]
}
 list_response(ual2, "List of Unknown Attributes")
}
}
```

oid_valid

oid_valid (*oid*)

This function returns `1` (True) if the object identified by *oid* is valid, such that the object still exists and oid !=oid_null() .

oid_visible_change_tracking

`0|1` = **oid_visible_change_tracking**(*oid*[, *mode*])

This function determines if the element specified by *oid* is visible with respect to a given change tracking view.

- *oid* — The element being checked. `oid_visible_change_tracking` returns `0` or `1` if *oid* is not visible or is visible, respectively, when the document is shown in the given change tracking view.

If *oid* is invalid, `oid_visible_change_tracking` returns `0`. If the document containing *oid* does not contain change tracking markup, `oid_visible_change_tracking` returns `1`.

- *mode* — Optional. Specifies a particular change tracking view. Valid values include:
  - `0` — `Original`
  - `1` — `Changes Applied`
  - `2` — `Changes Highlighted`

  If *mode* is omitted or any other value is supplied, the current change tracking view mode is used.

oid_write_graphic

`oid_write_graphic` (*`oid`*, *`path`*[, *`type`*[, *`attrmask`*[, *`quality`*[, *`cgmprofile`*]]]])

This function creates a graphic file based on the element identified by *oid* in the file format specified by *type* and writes the output file to the specified *path*. The *attrmask* parameter indicates how the graphic element should be scaled. The *quality* parameter can be used for JPEG graphics to better control the output file size balanced against output quality. The *cgmprofile* parameter can be used for CGM graphics to specify the CGM profile for the graphic.

A relative *path* is relative to the current working directory.

If *oid* refers to an intelligent graphic, the *type* can be `eps`, `jpg`, `png`, or `cgm`. If the *type* is set to `cgm`, the *cgmprofile* parameter can be used to specify the CGM profile for the graphic.

The settings of the `reproDepth` and `reproWidth` attributes on *oid* are used to create the graphic file. If none of these attributes are specified, the default values are used.

For graphics other than intelligent graphics, *type* can be `"gif"`, `"png"`, `"jpg"`, or `"jpeg"`. If *type* is omitted, the function will convert the element to GIF format.

attrmask can have the following values:

- `0` — Indicates the graphic should be scaled exactly as displayed in the Edit pane.

  This is the default setting.
- `1` — Indicates the graphic should be scaled based on the screen resolution.

  Any explicit scaling, such as that set in attribute values, is ignored for this setting.

If the graphic is an intelligent graphic, *attrmask* is ignored.

You can specify *quality* for a JPEG graphic. Set the value to a number from 1 to 100, where 1 is lowest quality and smallest file size and 100 is highest quality and largest file size. You may need to experiment with this setting for best results. The default is `80`. If *type* is `"gif"` or `"png"` or if the graphic is an intelligent graphic, any value specified for *quality* is ignored.

You can specify the *cgmprofile* for a CGM graphic. If *cgmprofile* is not set, then the value of the `set cgmprofile` preference is used. If that preference is also not set, the WebCGM 2.0 profile is used. The following CGM profiles are supported:

- `WebCGM_2.0`
- `WebCGM_1.0`
- `ISO_8632_:_1999`
- `ATA_GREXCHANGE_V2.10`
- `ATA_GREXCHANGE_V2.9`
- `ATA_GREXCHANGE_V2.8`
- `ATA_GREXCHANGE_V2.7`
- `ATA_GREXCHANGE_V2.6`
- `ATA_GREXCHANGE_V2.5`
- `ATA_GREXCHANGE_V2.4`
- `ATA_GREXCHANGE_V2.5/IsoDraw`
- `CALS_MIL-D-28003A`
- `SAE_J2008`
- `Model_Profile_(ISO_8632:1992/Am.1:1994)`
- `ISO_ISP_12071-1_(FCG11)_(Draft)`
- `ISO_ISP_12071-2_(FCG23)_(Draft)`
- `ISO_ISP_12071-3_(FCG32)_(Draft)`
- `ISO_ISP_12071-4_(FCG33)_(Draft)`
- `S1000D_V2.3`
- `S1000D_V2.2`

This function returns `1` (True) if the conversion succeeds, and `0` (False) if it fails. If it fails, the function writes an error message to the $ERROR predefined variable on page 78.

The following example converts every graphic and equation in a document to GIF format:

```
$g=0
$e=0
```

```
$o=oid_first()
while (oid_valid($o)) {
 if (oid_is_graphic($o))
 {
 $gname = "c:\\temp\\graphic" .$g. ".gif"
 oid_write_graphic($o,$gname)
 $g = $g + 1
}
 if (oid_type($o)== 4)
 {
 $ename = "c:\\temp\\equation" .$e. ".gif"
 oid_write_graphic($o,$ename)
 $e = $e + 1
}
 $o = oid_forward($o)
}
```

oid_xpath_boolean

oid_xpath_boolean (*oid*, *expr*[, *nsOid*])

This function evaluates an XPath expression in the context of an OID and returns `1` (true) if *expr* is true, otherwise it returns `0` (false).

- *oid* — The object to use as the context node.
- *expr* — A valid XPath expression.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
  - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *nsOid* (optional) — The node used for resolving namespace prefixes in the XPath expression.

  The *nsOid* parameter is useful when you need to evaluate the same XPath expression against many XML documents. Those documents might use a different namespace prefix than the XPath expression. If you provide an OID where the namespace prefix is declared, its URI can be matched up with the document instances regardless of the namespace prefixes they used. For example, you might have a configuration file that uses XPath expressions to specify naming rules with namespace prefixes declared on its root element.

A typical boolean expression will use one of the XPath boolean operators (=, !=, <, >, >=, <=). For example,
`count(sect1)=0`

Non-boolean results are converted to boolean using standard XPath rules:

- The numbers zero (`0`) and NaN (not a number) are converted to `false`. Other numbers are converted to `true`.
- Node set expressions returning at least one node are converted to `true`. Empty node sets are converted to `false`.
- Strings are converted to `true` if they contain at least one character (including white space). Empty strings are converted to `false`.

> **Note**
>
> Because non-empty strings are converted to `true`, the string "false" and the expression "`string(false())`" are both converted to `true`.

To handle errors, call this function from within a `catch`.

oid_xpath_integer

`oid_xpath_integer` (*`oid`*, *`expr`*[, *`nsOid`*])

This function evaluates an XPath expression in the context of an OID. This function evaluates *expr* as a floating-point number, rounds to the nearest integer, and returns that integer.

- *oid* — The object to use as the context node.
- *expr* — A valid XPath expression that can be represented as an integer.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
  - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *nsOid* (optional) — The node used for resolving namespace prefixes in the XPath expression.

The *nsOid* parameter is useful when you need to evaluate the same XPath expression against many XML documents. Those documents might use a different namespace prefix than the XPath expression. If you provide an OID where the namespace prefix is declared, its URI can be matched up with the document instances regardless of the namespace prefixes they used. For example, you might have a configuration file that uses XPath expressions to specify naming rules with namespace prefixes declared on its root element.

Non-numeric results are converted to numbers using standard XPath rules:

- Strings are parsed as floating-point numbers.
- Boolean expressions are converted to `1` or `0`.
- Node set expressions are first converted to strings, then parsed. A run-time error occurs if the result cannot be represented as an integer.

To handle errors, call this function from within a `catch`.

oid_xpath_matches

`oid_xpath_matches` (*`oid`*, *`expr`*[, *`nsOid`*])

This function tests whether an OID matches an XPath pattern. *expr* follows the same form as XSLT pattern matching expressions (`xsl:template` match attribute). Returns `1` if *oid* matches *expr* and `0` if not. To handle errors, call this function from within a catch.

This function has the following parameters:

- *oid* — The object to test.
- *expr* — A valid XPath pattern.
- *nsOid* (optional) — The node used for resolving namespace prefixes in the XPath expression.

  The *nsOid* parameter is useful when you need to evaluate the same XPath expression against many XML documents. Those documents might use a different namespace prefix than the XPath expression. If you provide an OID where the namespace prefix is declared, its URI can be matched up with the document instances regardless of the namespace prefixes they used. For example, you might have a configuration file that uses XPath expressions to specify naming rules with namespace prefixes declared on its root element.

Examples:

```
oid_xpath_matches(oid_caret(), "para")
```

Returns `1` for any `para` element.

```
oid_xpath_matches(oid, "chapter/para[1]")
```

Returns `1` for the first `para` in any `chapter`.

oid_xpath_nodeset

oid_xpath_nodeset (*oid*, *arr*, *expr*[, *nsOid*])

This function evaluates an XPath expression in the context of an OID and returns the result as an array of OIDs. This function can be used to identify a group of elements to be iterated through. It can also be used for navigation, for example, by passing one of the returned OIDs to `goto_oid`.

- *oid* — The object to use as the context node.
- *arr* — An unordered array of OIDs returned by *expr*.
- *expr* — A valid XPath expression that evaluates to a node set containing only XPath nodes that have associated OIDs.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
  - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *nsOid* (optional) — The node used for resolving namespace prefixes in the XPath expression.

  The *nsOid* parameter is useful when you need to evaluate the same XPath expression against many XML documents. Those documents might use a different namespace prefix than the XPath expression. If you provide an OID where the namespace prefix is declared, its URI can be matched up with the document instances regardless of the namespace prefixes they used. For example, you might have a configuration file that uses XPath expressions to specify naming rules with namespace prefixes declared on its root element.

This function fills the array *arr* with the first OID of each XPath node returned by *expr*. The return value is the number of OIDs stored in *arr*. Duplicate OIDs may be returned if more than one XPath node resolves to the same OID. `oid_null` on page 453 is returned for any XPath node that cannot be converted to an OID (for example, namespace nodes and attributes).

To handle errors, call this function from within a `catch`.

oid_xpath_string

oid_xpath_string (*oid*, *expr*[, *nsOid*])

This function evaluates an XPath expression with respect to an OID and returns the result of *expr* converted to a string.

- *oid* — The object to use as the context node.
- *expr* — A valid XPath expression.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
  - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *nsOid* (optional) — The node used for resolving namespace prefixes in the XPath expression.

  The *nsOid* parameter is useful when you need to evaluate the same XPath expression against many XML documents. Those documents might use a different namespace prefix than the XPath expression. If you provide an OID where the namespace prefix is declared, its URI can be matched up with the document instances regardless of the namespace prefixes they used. For example, you might have a configuration file that uses XPath expressions to specify naming rules with namespace prefixes declared on its root element.

Non-string results are converted to strings using standard XPath rules:

- Boolean expressions are converted to the string "`true`" or "`false`"
- Node set expressions return the string value of the first node, or an empty string if the node set is empty.
- Numbers are not specially formatted.

To handle errors, call this function from within a `catch`.

open

open(*path*[, *access*])

This function opens the file specified by *path* and returns an identifier that may be used in subsequent calls to `getline`, `put`, `close`, and other I/O functions. The path name is expanded using `expand_file_name` to resolve directory references and environment variables.

The *access* argument determines how the file is to be accessed. It must have one of the following values:

- `r` — open for reading only. The file must already exist.
- `w` — open for writing only. The file is created if it does not exist or truncated if it does.
- `a` — open for writing only. The file is created if it does not exist; if it does, then the pointer is positioned so that new output is added to the end of the file.
- `r+` — open for reading and writing. The file must already exist.
- `w+` — open for reading and writing. The file is created if it does not exist or truncated if it does.
- `a+` —open for reading and writing. The file must already exist and the pointer is positioned so that new output is added to the end of the file.

In addition, the following character may be appended to any of the *access* strings:

- `b` — open in binary mode. This is required to read/write binary files using the `read` and `write` functions. If `b` is not specified, then characters read from a file will be converted from the system character set to Unicode and vice-versa on output.

The default for *access* is `r`.

If a file is opened for both reading and writing, then a `seek` must be used between a `read` and a `write` operation.

There is a limit of, at most, 20 open file identifiers. The number may be less depending on the operating system.

If the file cannot be opened, `open` returns `-1`.

The `open` function also supports the special path names allowed for the *output*= option of the `eval` and `show` commands. If the first character of the output file starts with a question mark `?`, then the file name is actually a variable name whose value is set to the output written to the file identifier. If the second character is >, then the output is appended to the current value of the variable instead of replacing it. For example,

```
fid = open("?OUTPUT", "w")
```

returns a file identifier that can be used with `put` to write to a variable. The `open` function also supports the `"*"` (asterisk) output specification to write to the message window, thus

```
fid = open("*", "w")
```

returns a file identifier that can be used with `put` to send output to the message window. These extensions allow a generic output function to be written that can write to a file, variable or message window.

open_accept

open_accept (*ch*)

This function opens a network connection to a client using the channel identifier *ch* returned by a previous call to `open_listen`. `open_accept` returns `-1` on failure and sets the predefined variable `api_error` o an error message describing the error.

`open_accept` is normally called from the callback established by `open_listen`, which responds to an open notification for the listening channel. Otherwise, `open_accept` locks until a connection is made.

By default, the channel is opened in a blocking mode. This means that a `read` from the channel blocks until the operation completes. `channel_set_callback` can be used to change the channel so that subsequent reads (and writes) can be non-blocking.

The original listening channel remains open.

See `open_listen` for an example.

open_connect

open_connect (*host*, *port*)

This function opens a network channel to a TCP service on the specified host. *host* is the name of the host to connect to or its IP address. *port* is the integer port number to connect to (for example, 25), or the name of the service desired, such as, “smtp”.

`open_connect` returns a channel identifier on success that can be used in subsequent calls to `read`, `write`, `close` and other channel functions. It returns `-1` on failure and sets the predefined variable api_error to an error message describing the error.

By default, the channel is opened in a blocking mode. This means that a `read` from the channel blocks until the operation completes. `channel_set_callback` can be used to change the channel so that subsequent reads (and writes) can be non-blocking.

The sample function that follows uses `open_connect` to fetch an HTML document from an HTTP server (well known port 80). For clarity, it does minimal error checking and does not deal with the MIME headers that are normally transmitted preceding the actual document. This example also reads from the channel using blocking reads. See `channel_set_callback` for an example using asynchronous reads, the preferred method to use.

Examples

```
function http_get(host, remotepath, lclfile)
{
```

```
 local of = open(lclfile, "wb")
 if (of < 0) {
 message "http_get: couldn't open $lclfile for write"
 return 0;
}
 local ch = open_connect(host, 80)
 if (ch < 0) {
 message "http_get: $main::api_error"
 return 0;
}
 write(ch, "GET " . remotepath . " HTTP/1.0\r\n\r\n")
 local buf, len
 while ((len = read(ch, buf, 512)) > 0) {
 write(of, buf, len)
}
 close(ch);
 close(of)
 return len == 0
}
```

open_listen

open_listen (*port*, *funcname*[, *data*])

This function creates a channel to listen for incoming connections to implement a TCP service. `open_listen` returns a channel identifier on success that can be used in subsequent calls to `open_accept`. It returns `-1` on failure and sets the pre-defined variable api_error to an error message describing the error.

port is the integer port on which to listen (for example, 21) or the name of the service provided, such as, “ftp”. *funcname* is the name of the function that can be called when a connection becomes available and must be declared as: `function` *`funcname`*`(`*`ch`*`,` *`op`*`[,` *`data`*`])` where *ch* is the channel identifier returned by `open_listen` and *op* is an integer specifying the type of notification (which is always 0). This means that a connection is pending. *data* is the value of the user data argument passed to `open_listen`.

The callback function must call `open_accept` to create a channel for the new connection. As with the underlying Berkeley socket function, `listen`, `open_listen` limits the number of pending connections (the backlog) to 5.

The channel created by `open_listen` is non-blocking. It remains open until explicitly closed by calling `close`.

The example of the simple server that follows accepts and executes Arbortext Editor commands over a network channel. It uses a simple protocol where each incoming command is delimited by a **CTRL+A** character (“\001”) since commands may include embedded new lines. After executing the command, the server writes back the value of the predefined variable status as a string, again terminated by **CTRL+A**. The client should read the reply before sending another command.

```
package server
RE="\001"; # used as a message delimiter
function listen(port) {
 local ch = open_listen(port, 'server::doaccept')
 if (ch < 0) {
 message "server::listen: $main::api_error"
}
 return ch
}
function doaccept(ch, what) {
 if (what == 0) {
 local fid = open_accept(ch)
 if (fid < 0) {
 message "server: $main::api_error"
 close(ch);
}
 # set the connection to non-blocking mode
 channel_set_callback(fid, 'server::notify')
}
}
function async_read(ch) {
 local cmd, len
 len = read(ch, cmd, 512, RE)
 if (len == -2) {
 return; # incomplete read
}
 if (len <= 0) {
 if (len == -1) {
 message "server: $main::api_error"
}
 close(ch)
 return
}
 chop(cmd); # remove RE
 execute(cmd); # in context of current_doc()
 # send back reply containing $status for command
 write(ch, "=" . main::status . RE)
}
# Callback for I/O on server connection
function notify(ch, what) {
 switch(what) {
 case 0: # open
 break;
 case 1: #close
 close (ch);
 break;
 case 2: # read
 async_read(ch)
 break;
}
}
```

option

option(*string*[, *session*])

This function returns the local scope (window, document, or view) value of the `set` option specified by *string*, which may be an initial substring. If the optional boolean *session* parameter is true, the option returns the session scope value for the `set` option specified by *string*. The result is a string which may be passed to the `set` command. Boolean values return `on` or `off`.

Examples

```
$ptr = option("printer");
$cxon = option("contextrules") == "on";
if (option("tabletags",1) == "off") { ...}
```

option_path_list

option_path_list (*name*, *arr*)

This function returns in array variable *arr* the value of the path-list set option *name*. It returns the number of items placed in array variable *arr*. If *name* is not a path-list set option, or the set option has a null (empty) value, it returns 0.

Note

A path-list option refers to set options such as `javaclasspath` and `catalogpath`, whose value is a semicolon-separated list of paths to directories or files.

Values are returned by splitting the list in the set option into separate file paths and returning one path in each array element. For example, if set option `catalogpath` has the value `a;b;c`, where a, b, and c are each absolute paths to directories containing document type information, this function will return `a` in *arr[1]*, `b` in *arr[2]*, and `c` in *arr[3]*. The function itself will return `3` in this case.

option_persists

option_persists (*option*)

This function returns `1` if the value of the preference specified in *option* automatically persists across sessions (written to the `arbortext.wcf` preferences file on exit). The function return `0` if the **Preferences** dialog box must be used to save *option*. The function also returns `0` if *option* is not a valid preference.

option_scope

option_scope (*option*)

This function returns the scope for the specified *option*. Available return values are:

- `global` — specified *option* has no local scope, only a session scope.
- `window` — specified *option* has a session scope and a local scope of window.
- `doc` — specified *option* has a session scope and a local scope of document.
- `view` — specified *option* has a session scope and a local scope of view.

If the specified *option* is invalid, the function returns nothing.

Examples

```
eval option_scope("tagdisplay")
$ret = option_scope("showentities")
eval option_scope("gentext")
```

option_set

option_set (*name*, *value*)

This function sets the global-scope set option *name* to the value *value*.

option_type

option_type (*option*)

This function returns the type of the specified *option*. Possible return values are:

- `string` — type is a string.
- `token` — type is a token.

  Similar to `string`, except its value should not be quoted in a `set` command.
- `file` — type is a file name.
- `pathlist` — type is a path list (a list of file paths separated by a semicolon).
- `bool` — type is a boolean (`on` or `off`).
- `number` — type is an integer.
- `float` — type is a floating point number.
- `color` — for options that specify foreground colors, type is a color (named color or a RGB value).
- `bgcolor` — for options that specify background colors, type is a color (named color or a RGB value).

- `dimen` — type is a dimension (value and a unit).

  Valid units are `in`, `cm`, `mm`, `pc`, `pi`, `pt`, `em`, and `px`.
- `percent` — type is a dimension like `dimen`, but the unit may also be `%`.
- `enum` — type is an enumerated type.

  Call the `option_values` function on page 477 for a list of its valid values.
- `num_enum` — type is a combination of an integer and an enumerated type.

  Call `option_values` function for a list of its valid enumerated values.

  Call `option_value_min` on page 476 and `option_value_max` on page 476 functions for the range of valid number values.
- `hook` — type is a hook function.

If the specified *option* is invalid, then the function returns `invalid`.

Examples

```
eval option_type("tagdisplay")
eval option_type("gentext")
option_type("showentities")
```

option_value_max

option_value_max (*option*)

This function returns the maximum allowable value for the specified *option*.

If the specified *option* is invalid, or is not numeric, then the function returns `-1`.

Example

```
option_value_max("fontpercent")
```

If the option's value is expressed in units, you can return the kind of units with `option_value_units` on page 477.

option_value_min

option_value_min (*option*)

This function returns the minimum allowable value for the specified *option*.

If the specified *option* is invalid or is not numeric, then the function returns `-1`.

Example

```
option_value_min("outputrecordlength")
```

If the option's value is expressed in units, you can return the kind of units with `option_value_units` on page 477.

option_value_units

option_value_units (*option*)

This function returns the units being used by the specified *option*.

If the specified *option* is invalid or doesn't use units, then the function returns an empty string.

The following example obtains the unit of measure for the `set docmapwrapwidth` on page 788 command option.

```
option_value_units("docmapwrapwidth")
```

option_value_validate

option_value_validate(*option*, *value*)

This function validates the specified *value* for the specified *option*.

If the specified *value* is valid, the function returns `1`. If the specified *value* is invalid, the function returns `0` and an explanation in an error dialog box.

Example

```
option_value_validate("tagdisplay", "full")
option_value_validate("tagdisplay", "notvalid")
```

option_values

option_values (*option*, *arr*)

This function fills the array *arr* with a list of permissible values for the specified *option* and returns the number of entries in the array.

If the specified *option* is invalid or is not an enumerated type, then the function returns `0`.

Example

```
option_values("tagdisplay", $arr)
```

ord

ord(*value*)

This function returns integer value of first character in string value of *value*. If the character is a Unicode character, the numeric sixteen-bit number is returned. For example the value of `ord("a")` is `97`.

pack

pack(*template*, ...)

This function packs one or more values into a binary data structure, returning a byte string containing the structure. This is very similar to the Perl function of the same name. The *template* parameter is a sequence of characters that specify the type of each value, as follows:

- `a` — an ASCII string, null padded
- `A` — an ASCII string, blank value
- `c` — a signed character value
- `C` — an unsigned character value
- `d` — a double-precision floating point number in native format
- `f` — a single-precision floating point number in native format
- `i` — a signed integer (32–bit) value
- `I` — an unsigned integer (32–bit) value
- `l` — a signed long value
- `L` — an unsigned long value
- `n` — a short integer value in network (big-endian) order
- `N` — a long (32–bit) value in network (big-endian) order
- `p` — a pointer to a null terminated string
- `s` — a signed short (16–bit) value
- `S` — an unsigned short (16–bit) value
- `x` — a null byte
- `X` — back up a byte
- `z` — a Unicode string in big-endian (network) order, null padded
- `Z` — a Unicode string in little-endian order, null padded
- `@` — null fill to absolute position

Each character may be followed by a number that specifies a repeat count. The character and repeat count comprise a field specifier. Field specifiers may be separated by white space. For all type specifiers except `a`, `A`, `x`, `X`, and `@`, the repeat count specifies how many values from the list of arguments are to be used. For example:

```
pack("s2", 1, 2)
```

packs values `1` and `2`, into a structure of 2 shorts. If the repeat count is *, then all remaining values are used, for example:

```
pack("i*", 3, 4, 5, 6)
```

produces a structure of four integers.

The repeat code for `z` and `Z` is the number of Unicode characters. Thus, `z4` results in 8-bytes written to the packed string. To get a blank-padded Unicode string, use:

```
pack("z20", ustr . dupl(" ", 20 - length(ustr)))
```

For specifiers `a` and `A`, only a single value is used. The repeat count specifies the size of the output string, padded with nulls if `a`, or space if `A` was given. The type specifiers `x`, `X`, and `@` do not use up any values. The repeat count for `x` and `X` specify how much of the output string should be expanded or contracted. `@` is similar to `x` but the repeat count specifies an absolute instead of relative position.

Since Arbortext Editor does not support floating point as a basic type, the values corresponding to the `f` and `d` type specifiers must be written as strings. For example:

```
pack("df", "1.5e-4", "72.27")
```

The binary representation of real numbers is the native machine format which is not portable.

If an argument specifies an array name, then all the elements of the array from 1 to `high_bound(` *`array name`* `)` are used if the template specifies enough fields. For example,

```
split("1 2 3 4 5", array)
split("1 2 3 4 5", array)
n5 = pack("i*", arry)
```

generates a structure of 5 integers. Currently, there is a limit of 250 field values.

The following three examples generate a string of four characters (and are equivalent):

```
pack("cccc", 97, 98, 99, 100)
pack("c4", 97, 98, 99, 100)
pack("a4", "abcdefg")
```

Note that in the latter case, only the first four characters of the value are used.

The example that follows involves writing a port number in network order to a network channel (possibly connected to a non-Intel based machine):

```
write(ch, pack("n",portno))
```

If the other end was a Arbortext Editor client, `unpack` would be used to convert the port number. For example:

```
 read(ch, buf, 2)
portno = unpack("n", buf)
```

`pack` may also be used to preallocate a string to a particular size, for example,

```
str = pack("x1000");str = ''
```

creates a string of 1000 bytes and gives it a null initial value. This is useful in cases where a large string is created by appending many small strings using the concatenation assignment operator .= in a loop. Preallocating the string to a size close to the final size reduces the number of times Arbortext Editor has to reallocate the string as characters are appended. This also reduces the execution time needed for the loop.

To pack a Unicode character, use `s` or `n`. For example,
`pack("s2", ord("a"), chr(0x2665))`

The `a` and `A` designators convert a Unicode string into a possibly multi-byte string in the system character set. Note, if a field length is specified, for example, `a7`, this could result in a partial character being stored if the system character set is a multi-byte character set.

If you are converting a Unicode string parameter to an 8-bit string for passing to a dynamic library, it is best to pass the string as a parameter to the `dl_call` function and let `dl_call` to the conversion for you. One case in which it is necessary to use `pack` is if the parameter is an output parameter and one needs to allocate enough space to hold the return value. Then the ACL code will need to use `pack` with something like `"a2000"` to allocate a big enough string to hold the result. Note that this assumes that the DLL is an 8-bit DLL and conversion is necessary.

package (Function)

package(*package*)

This function changes the current package to the name given by *package* which must specify a package previously defined by the `package` command. The function returns the name of the package previously in effect on success. It returns a null string if *package* does not exist.

Unlike the `package` command, the package name is not evaluated until the `package` function is executed.

package_file

package_file ([*package*])

This function returns the path name associated with package *package,* or the current package if no argument is specified. Note that if more than one command file uses *package*, the first file loaded that defines the package with the *package* command is considered the package file.

package_name

package_name ([*level*])

This function returns the name of the current package. If the argument *level* is given, it specifies the number of call frames to go back before the current one to obtain the package name For example, `package_name` returns the name of the package active at the time of the call to function containing the call to `package_name`.

packages

packages (*arr*)

This function fills the associative array *arr* with a list of all loaded packages. It returns the number of loaded packages. The package name is used as the key and the associated file name is the value of each array element.

panel_popup

panel_popup (*name*)

This function displays the predefined dialog box named *name* and is one of the following strings:

- `bookmarks` — the **Bookmarks** dialog box (from **Insert ▸ Insert Bookmark**)
- `hyperlink` — the **Create Hyperlink** dialog box (from **Tools ▸ Create Hyperlink**; this command is available only if the `set hyperlinkmenus` on page 832 command is set to `on`.
- `macros` — the **Macros** dialog box (from **Tools ▸ Macro ▸ Macros**)
- `record` — the **Record Macro** dialog box (from **Tools ▸ Macro ▸ Record New Macro**)
- `symbols` — the Windows **Insert Symbol** dialog box (from **Insert ▸ Symbol**)
- `tagtemplates` — the **Tag Templates** dialog box (**Tools ▸ Tag Templates**)

paragraph_tag_name

paragraph_tag_name (*doc*)

This function returns the name of the primary paragraph tag specified in the `.dcf` file for the document type associated with *doc*. This tag is inserted when users select the paragraph button on the Application toolbar.

The function returns the null string if there isn't a paragraph tag designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

path_doc

path_doc (*path*)

This function searches the list of open documents for the path name given by *path*, which may be the name of a document directory or a plain file, and returns the corresponding document identifier if found, or `-1` if not. If path does not specify an absolute path name, the current directory is prepended.

Examples

This function can be written using other built-ins as follows:

```
function path_doc(name) {
local i, n, L[], path
 path = absolute_file_name(name);
 if (file_directory(path)) {
 path .= "/" . basename(name) . ".sgm"
}
 n = doc_list(L)
 for (i = 1; i <= n; i++) {
 if (path == doc_path(L[i])) {
 return L[i];
}
}
 return -1;
}
```

path_public_ids

path_public_ids (*path*, *arr*[, *first_only*[, *catpath*]])

This function returns an array of public ids mapped to the specified path PUBLIC and DTDDECL catalog entries. The function returns the number of elements in the array. If *first_only* is non-zero, then only the first match (if found) is returned. Otherwise, all public ids mapped to the path are returned (the default). If *catpath* is not specified, then the value of the option *catalogpath* is used for the list of catalog files to search. Otherwise, *catpath* specifies an alternate path list to search.

perlscript

perlscript (*expr*[, *window*])

This function sends the string *expr* to the Microsoft Windows PerlScript interpreter to be evaluated, returning the result as a string. If the *window* parameter is not specified, the PerlScript expression is evaluated in the global PerlScript script context so previously defined PerlScript functions are accessible.

The optional *window* parameter is the identifier of a XUI dialog box that has a script context. If *window* is provided, the script executes in that dialog box's script context.

This function requires that a PerlScript interpreter be installed.

Example:

```
name = perlscript('$Application->prompt("Name", "")')
```

Note

In the Microsoft Script Engine interfaces, “True” is represented by one (`1`) and “False” by zero (`0`).

paths_equal

`paths_equal` (*`path1`*, *`path2`*)

Determines if the two given paths actually represent the same file or directory. This will allow “subst” and “network mapped” drives to obtain the final target path for purposes of comparison.

Returns non-zero if the paths represent the same file or directory. Otherwise, `paths_equal` returns `0`.

preference

`preference` (*`string`*[, *`factory`*])

This function returns the preference scope value of the `set` option specified by *string*, which may be an initial substring. This value is the value of the `set` option when Arbortext Editor is first started, which is also the `.associated` setting in the **Preferences** dialog box.

Set the optional boolean *factory* parameter to true to return the default value for the `set` option specified by *string*. The result is a string which may be passed to the `set` command.

Note

Boolean values return `on` or `off`.

Examples

```
$tgfntclr = preference("tagfontcolor");
if (preference("pagebreaktext",1) == "off") { ...}
```

Note

Not all `set` options are preferences. If the *string* parameter specifies a `set` option that is not a preference, the return value will be empty.

procins_tag

procins_tag (*name*[, *doc*])

This function returns 1 (True) if the name specified by *name* is an SGML processing instruction. The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

profile_alias

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

Returns the alias name for the specified profile attribute.

- *attr* — The attribute for which the alias name is returned. If no such alias exists, null is returned.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_aliases

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

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

- *arr* — An array of the profile aliases defined in the configuration file associated with the profiling session. profile_aliases returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_allowed

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.

- *alias* — The profile to apply.
- *oid* — The element to be profiled.

profile_attr

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

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

- *alias* — The alias for which the name of the profile attribute is returned. If no such attribute exists, `null` is returned.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_attrs

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

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

- *arr* — An array of the profile attributes defined in the configuration file associated with the profiling session. `profile_attrs` returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_class_values

profile_class_values (*arr*, *profile*[, *doc*])

Note

`profile_class_values` is deprecated.

This function fills the array *arr* with a list of allowed values that are defined for the given *profile* attribute in the document type configuration file (`.dcf`) for the document type associated with *doc*.

The function returns the allowed values in the order in which they are declared in the `.dcf` file. This function also returns the number of entries in the array. The first entry is stored at index 1.

profile_classes

profile_classes (*attributeArr*, *aliasArr*[, *doc*])

Note

`profile_classes` is deprecated.

This function fills the arrays *attributeArr* and *aliasArr* with a list of the profile attributes and their aliases declared in the document type configuration file (`.dcf`) for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

This function returns the array values in alphabetical order. It also returns the number of entries in each array. The first entry is stored at index 1.

profile_config

`profile_config` (`[`*`doc`*`]`)

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

- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used. If *doc* is invalid or no configuration file is defined for profiling, `profile_config` returns `-1`.

profile_conflictshadingbackground

`profile_conflictshadingbackground`(`[`*`doc`*`]`)

Returns the conflict color set for the profile for a particular document.

- *doc* — Optional. The target document . If *doc* is not supplied, the current document is used.

The function returns a color name or a color number in the form `#rrggbb`. If no color is defined for the profile, an empty string is returned.

Note

While 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_default_value

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

Returns the default value for the specified RADIO_PROFILE type node.

- *alias* — The alias of the RADIO_PROFILE type node for which the default value is returned. If *alias* is not a RADIO_PROFILE type node or if the default value has not been set, profile_default_value returns an empty string .
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used. If *doc* is invalid or no configuration file is defined for profiling, profile_config returns -1.

profile_default_value_node

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

Returns the default value profilenode for the specified RADIO_PROFILE type node.

- *alias* — The alias of the RADIO_PROFILE node for which the default value is profilenode returned. If *alias* is not a RADIO_PROFILE type node or if the default value has not been set, profile_default_value returns profilenode_null.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used. If *doc* is invalid or no configuration file is defined for profiling, profile_config returns -1.

profile_element_allowed

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

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

- *alias* — The alias of the profile to be applied.
- *tagname* — The element to be profiled.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_element_attr_tests

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 (if the *not_indicator* of `profile_elements_list` is `0`) or not applied to (if *not_indicator* is `1`).

- *alias* — The alias of the profile to be applied (or not applied).
- *tagname* — The element to be profiled (or not profiled).
- *arr* — An array of the attribute name(s) and value(s) for *tagname* that *alias* could be applied to or not applied to. `profile_element_attr_tests` returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_elements_list

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

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

- *alias* — The alias of the profile to be applied (or not applied).
- *arr* — An array of elements to which *alias* can (or cannot) be applied. `profile_elements_list` returns the count of the array *arr*.
- *not_indicator* — When set to `0`, specifies that *alias* can be applied to the elements listed in *arr*. When set to `1`, specifies that *alias* cannot be applied to the elements listed in *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_is_foldered

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

Returns `1` if the specified profile class is a `FOLDERED_PROFILE` class. Otherwise, `profile_is_foldered` returns `0`.

- *alias* — The profile class to be evaluated.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_is_radiochoice

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

Returns 1 if the specified profile class is a RADIO_PROFILE class. Otherwise, profile_is_radiochoice returns 0.

- *alias* — The profile class to be evaluated.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_is_standard

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

Returns 1 if the specified profile class is a STANDARD_PROFILE class. Otherwise, profile_is_standard returns 0.

- *alias* — The profile class to be evaluated.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_resolution

profile_resolution (*oid*, *logical_expression*)

Returns 1 if the specified element would be included if the profile was resolved using the specified logical expression. Otherwise, profile_resolution returns 0.

- *oid* — The element to be evaluated.
- *logical_expression* — XML markup used for resolving the profile. The markup conforms to the structure of the <LogicalExpression> element in the profile configuration file.

profile_rootnode

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

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

- *alias* — The profile class for which the profilenode is returned.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is 0, the current document is used.

profile_rootnodes

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

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

- *arr* — An array of profilenodes identifiers of type `STANDARD_PROFILE`, `RADIO_PROFILE`, or `FOLDERED_PROFILE` for each profile defined in the profile configuration file. `profile_rootnodes` returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_shadingbackground

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

Returns the shading color set for a profile attribute in a particular document.

- *alias* — The alias of the profile attribute
- *doc* — Optional. The target document . If *doc* is not supplied, the current document is used.

The function returns a color name or a color number in the form `#rrggbb`. If no color is defined for the alias, an empty string is returned.

profile_type

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

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

- *alias* — The profilenode for which the type is returned. `profile_type` returns the following values:
  - `0` — *alias* is an `INVALID_PROFILE` type.
  - `1` — *alias* is an `STANDARD_PROFILE` type.
  - `2` — *alias* is an `RADIO_PROFILE` type.
  - `3` — *alias* is an `FOLDERED_PROFILE` type.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_valid

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

Returns `1` if the specified alias is a valid profile. Otherwise, `profile_valid` returns `0`.

- *alias* — The profile to be evaluated.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_value_node

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

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

- *alias* — The profile class containing *value*.
- *value* — The value for which the profilenode is returned.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_value_nodes

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

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

- *alias* — The profile class to be evaluated.
- *arr* — An array of allowed value profilenodes for the profile class *alias*. The array is populated in the order in which the values are specified in the associated profiling configuration file. `profile_value_nodes` returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_value_separator

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

Returns the value separator for the specified profile.

- *alias* — The profile for which the value separator is returned. The value separator is specified on the `<Profile>` element in the profile configuration

file. the separator separates multiple values from each other. The default value is a semicolon.

- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_values

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

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

- *alias* — The profile class to be evaluated.
- *arr* — An array of the allowed values for *alias*. The array is populated in the order in which the values are specified in the profiling session configuration file. `profile_values` returns the count of the array *arr*.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

profile_values_shadingbackground

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

This function populates the array *array[]* with the colors for the profile attribute for *alias* for document *doc*.

If is not supplied, the current document is used. If is supplied and has a non-zero value, the profile shading color for the Profile element is also included as the last element in the array.

The function returns the color names or color numbers in the form #rrggbb. If no color is defined for the profile value, an empty string will be in that array position. The array is filled in the same order as the function call profile_values() on page 492.

profilenode_ancestors

profilenode_ancestors(*profilenode*, *arr*)

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

- *profilenode* — The node for which the number of ancestors is determined.
- *arr* — An array of the profilenode identifiers of the folders that are ancestors of the node identified by *profilenode*. The list of values includes the

profilenode of type STANDARD_PROFILE, RADIO_PROFILE, or FOLDERED_PROFILE for that particular profile branch. The array is populated in the order in which the ancestors are listed in the profile configuration file, starting with the STANDARD_PROFILE, RADIO_PROFILE, or FOLDERED_PROFILE profilenode. profilenode_ancestors returns the count of the array *arr*.

profilenode_attr

profilenode_attr (*profilenode*)

Returns the name of the profile attribute for a profilenode.

- *profilenode* — The node for which the profiling attribute is returned.

profilenode_attr returns the name of the profile attribute for the node specified by *profilenode*. profilenode_attr returns the null string if no such attribute exists.

profilenode_children_nodes

profilenode_children_nodes(*profilenode*, *arr*)

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

- *profilenode* — The node for which the number of child nodes is returned.
- *arr* — An array of the profilenode identifiers of the nodes that are children of *profilenode*. profilenode_children_nodes returns the count of the array *arr*.

profilenode_default_value

profilenode_default_value(*profilenode*)

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

- *profilenode* — The node for which the default value is returned.

profilenode_default_value returns an empty string if *profilenode* does not pertain to a RADIO_PROFILE node, or if the default value has not been set in the configuration file.

profilenode_default_value_node

profilenode_default_value_node(*profilenode*)

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

- *profilenode* — The node for which the default value profilenode is returned.

`profilenode_default_value` returns an empty string if *profilenode* does not pertain to a `RADIO_PROFILE` node, or if the default value has not been set in the configuration file.

profilenode_element_allowed

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

Determines if an element can be profiled using a given profilenode. Returns `0` if the element cannot be profiled, or if either the element or profilenode is invalid.

- *profilenode* — The node defining the profiling to be applied.
- *tagname* — The element to be profiled.

`profilenode_element_allowed` returns `1` if *tagname* can be profiled using *profilenode* or if either *tagname* or *profilenode* is invalid.

profilenode_element_attr_tests

`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* — The node defining the profiling.
- *tagname* — The element to be profiled as defined in the `<ProfileElement>` or `<NotProfileElement>` elements in the profile configuration file.
- *arr* — An array of the attribute name(s) and value(s) for *tagname* that the profile represented by *profilenode* could be applied to or not applied to. Attribute names and values are specified on the `<AttributeTest>` element in the `<ProfileElement>` or `<NotProfileElement>` elements in the profile configuration file.

`profilenode_element_attr_tests` returns the count of the array *arr*.

profilenode_elements_list

`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* — The node defining the profiling.
- *arr* — An array of the elements at the profile identified by *profilenode* could (or couldn't) be applied to, as specified in the `<ProfileElement>` or `<NotProfileElement>` elements in the profile configuration file.
- *not_indicator* — If `0` the profile can only be applied to the list of element names. If `1` the profile can be applied to any element NOT in the list of element names.

`profilenode_elements_list` returns the count of the array *arr*.

profilenode_is_foldered

`profilenode_is_foldered`(*`profilenode`*)

Determines if the root node of a node is a `FOLDERED_PROFILE` node.

- *profilenode* — The node for which the type of root node is determined.

`profilenode_is_foldered` returns `1` if the root node of *profilenode* is a `FOLDERED_PROFILE` node. Otherwise, `profilenode_is_foldered` returns `0`

profilenode_is_radiochoice

`profilenode_is_radiochoice`(*`profilenode`*)

Determines if the root node of a node is a `RADIO_PROFILE` node.

- *profilenode* — The node for which the type of root node is determined.

`profilenode_is_radiochoice` returns `1` if the root node of *profilenode* is a `RADIO_PROFILE` node. Otherwise, `profilenode_is_radiochoice` returns `0`

profilenode_is_standard

`profilenode_is_standard`(*`profilenode`*)

Determines if the root node of a node is a `STANDARD_PROFILE` node.

- *profilenode* — The node for which the type of root node is determined.

`profilenode_is_standard` returns `1` if the root node of *profilenode* is a `STANDARD_PROFILE` node. Otherwise, `profilenode_is_standard` returns `0`

profilenode_name

profilenode_name (*profilenode*)

Returns the profile alias, folder name, or profile value of *profilenode* as follows:

- Returns the profile alias if *profilenode* is a STANDARD_PROFILE, RADIO_PROFILE, or FOLDERED_PROFILE node.
- Returns the folder name if *profilenode* is a PROFILE_FOLDER node.
- Returns the profile value if *profilenode* is an ALLOWED_VALUE node.

An empty string if *profilenode* is invalid.

profilenode_parent

profilenode_parent (*profilenode*)

Returns the profilenode of the immediate ancestor of a profilenode.

- *profilenode* — The node for which the immediate ancestor is returned.

profilenode_null is returned if *profilenode* has no parent or is invalid.

profilenode_rootnode

profilenode_rootnode (*profilenode*)

Returns the top-level or root node of a profile class.

- *profilenode* — The node having the profile class for which the top-level node is returned.

profilenode_rootnode returns a profilenode of type STANDARD_PROFILE, RADIO_PROFILE, or FOLDERED_PROFILE for the profile class identified by *profilenode*. profilenode_null is returned if no top-level node exists.

profilenode_shadingbackground

profilenode_shadingbackground(*profilenode*)

Returns the color for the profile node *profilenode*.

The function returns the color name or color number, in the form #rrggbb, of the node or of an ancestor of the node if the color is not found on that node. If no color is defined for , or if the parameter is invalid, an empty string will be returned.

profilenode_type

profilenode_type (*profilenode*)

Returns a value based on the *profilenode* TYPE as follows:

- 0 — Returned if TYPE is an INVALID_PROFILE node.
- 1 — Returned if TYPE is an STANDARD_PROFILE node.
- 2 — Returned if TYPE is an RADIO_PROFILE node.
- 3 — Returned if TYPE is an FOLDERED_PROFILE node.
- 4 — Returned if TYPE is an PROFILE_FOLDER node.
- 5 — Returned if TYPE is an ALLOWED_VALUE node.

profilenode_valid

profilenode_valid (*profilenode*)

Returns 1 if *profilenode* is a valid profilenode identifier. Otherwise, profilenode_valid returns 0.

profilenode_value_nodes

profilenode_value_nodes(*profilenode*, *arr*)

Returns the number of value profilenodes for a node.

- *profilenode* — The node for which the number of value profilenodes is returned.
- *arr* — An array of the allowed values profilenodes for *profilenode*. The array is populated in the order in which the values are specified in the profile configuration file.

profilenode_value_nodes returns the number of profilenodes stored in the array *arr*.

profilenode_value_separator

profilenode_value_separator(*profilenode*)

Returns the value separator corresponding to the specified node.

- *profilenode* — The node for which the value separator is returned.

The value separator is specified on the <Profile> element in the profile configuration file. It is used to separate multiple values from each other. The default value for it is semicolon.

profilenode_values

profilenode_values (*profilenode*, *arr*)

Returns the number of allowed values for the specified node.

- *profilenode* — The node for which the number of allowed values is returned.
- *arr* — An array of the allowed values for *profilenode*. The array is populated in the order in which the values are specified in the profile configuration file.

`profilenode_values` returns the number of values stored in the array *arr*.

progressbar_available

progressbar_available()

Returns 1 if the progress bar dialog is currently available. If the function returns 0, all other `progressbar_*` functions (progressbar_visible() on page 500, progressbar_start_job() on page 499, progressbar_update() on page 499, progressbar_cancelled() on page 498, progressbar_close() on page 498) will fail silently.

The progress bar dialog will not be available if the environment is running in Publishing Engine or on a non-Windows platform.

progressbar_cancelled

progressbar_cancelled()

Returns 1 if the **Cancel** button on the progress bar dialog box has been activated. Returns 0 if not, or if there is no current progress bar dialog box.

Note

This function does not close the progress bar dialog box. The caller is responsible for invoking progressbar_close() on page 498 if this function is used to detect job cancellation.

progressbar_close

progressbar_close()

Closes the progress bar dialog box. This function is the only way to dismiss the progress bar dialog box. It cannot be closed by the user. For this reason, it is important that scripts that use progressbar_start_job() on page 499 also call this function before exiting.

progressbar_start_job

progressbar_start_job(*jobLabel*, *taskLabel*[, *nTasks=0*[, *allowCancel=1*[, *delay=0*]]])

Initializes a new job to be reported in the progress bar dialog box. If the progress bar dialog boxis already being shown, the new job replaces any existing job. If the dialog box is not currently visible this function raises it, after a delay if *delay* has a non-zero value.

- *jobLabel* — a short description of the overall job. This will be shown in bold text above the progress bar in the dialog box.
- *taskLabel* — a short description of the initial task. This will be shown in regular text below the progress bar in the dialog box.
- *nTasks* — the total number of steps required by the job. If set to `0` (the default) or a negative value, the progress bar will be shown in indeterminate, or waiting, mode.
- *allowCancel* — whether to enable the **Cancel** button on the dialog box. The default is 1 (enable).

  Operations using the progress bar API that do not regularly check the result of progressbar_cancelled() on page 498 should disable the **Cancel** button.
- *delay* — the number of seconds to wait before showing the progress bar dialog box.

  Some jobs may execute quickly or slowly depending on various features, and it may not be desirable to display the progress bar dialog immediately. When the dialog box is shown after the delay, its state will reflect the most recent call to progressbar_update() on page 499.

> **Note**
>
> If progressbar_close() on page 498 is called before the delay expires, the dialog box will not be shown.

It is important that scripts that use this function also call `progressbar_close()` before exiting, since the progress bar dialog box cannot be dismissed by the user..

progressbar_update

progressbar_update (*curTask*[, *taskLabel=""*[, *nTasks=-1*]])

Updates the state of the progress bar dialog box.

Note

If this function is called either without progressbar_start_job() on page 499 being called first, or after progressbar_close() on page 498 has been called, it has no effect.

- *curTask* — the current task index. This value should be less than or equal to *nSteps*, if specified, or less than or equal to the value passed to the most recent call to `progressbar_start_job()`.
- *taskLabel* — a short description of the current step. If the empty string is specified, the step label field is left unchanged. This is the default. To specify an empty step label, use a single space (`" "`).
- *nTasks* — the number of known steps in the current job. The known number of steps may change during the course of a job — using this parameter allows programs to update the number of steps as the job progresses. A negative value leaves the number of steps reflected by the progress bar unchanged. A value of `0` puts the progress bar into indeterminate, or waiting, mode.

progressbar_visible

`progressbar_visible` `()`

Returns 1 if the progress bar dialog is currently visible, 0 otherwise.

public_id

`public_id` `([`*`doc`*`])`

This function returns the SGML public identifier for the document being edited, but it returns the null string for an ASCII file. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

public_id_path

`public_id_path` `(`*`pubid`*`[, `*`unused`*`[, `*`catalogpath`*`]])`

This function returns the system path name for the SGML public identifier string *pubid*. If the identifier is not found in any of the `catalog` files in the search path given by the `set catalogpath` option, the `public_id_path` function returns a null string.

The second argument, *unused*, is no longer used. It should be the null string.

The *catalogpath* argument specifies a list of directories to search for the `catalog` file. If it is not specified, the path list specified in the `set catalogpath` option, which is initialized from the environment variable *APTCATPATH*, is searched.

put

put(*fid*, *str*)

This function writes the characters given by *str* to the file identified by *fid*, which must be a return value from a previous call to `open`. The file associated with *fid* must have been opened for writing.

- `put` does not add any newline characters after *str*. If linefeeds are desired, they must be given as part of the string. The output to the stream is buffered; `flush` can be used to force characters to be written.
- `put` returns the number of characters written or `-1` on failure (such as an attempt to write to a file identifier opened exclusively for reading).

pwd

pwd([*startup*])

This function returns the current working directory.

The flag argument *startup* is optional. If non-zero, the function returns the working directory at the time Arbortext Editor was started. If it is omitted or zero, `pwd` returns the current working directory.

qsort

qsort(*arr*[, *flags*])

The `qsort` function sorts the array *arr* in place according to the options specified by *flags*. It returns 1 on success, or 0 if the array is not a regular array (that is, an associative array).

flags is a bitmask that specifies sort options and is constructed by ORing the flags from the following list:

- `0x1` — ignore the case of the characters.
- `0x2` — reserved for future use.
- `0x4` — reverse the sense of the comparisons so the greatest value becomes the first array element.
- `0x8` — sort in numerical order. Bit 1 is ignored if this bit is set.

If *flags* is 0 or omitted, the sort is according to the collating sequence of the current locale.

quote

quote(*string*[, *flags*])

This function returns a copy of the string represented by *string* with embedded backslash (\), quote ("), and dollar sign ($) characters escaped with \ to form a valid ACL string operand. Each literal new line character in the string is replaced with the sequence \n. The resulting string is delimited with double quotes. For example, `quote('String with \ and $ embedded')` would return `"String with \\ and \$ embedded"`.

If the *flags* parameter is `0x01` then dollar signs are not escaped, allowing variable substitution to be performed on the resulting string.

read (Function)

read(*fid*, *buf*, *len*[, *re*])

This function reads up to *len* characters from the file or channel identified by *fid* into the scalar variable *buf*. This function returns the number of characters read, or `0` at the end of the file if the channel was closed. It returns `-1` if an error occurred. *fid* is a file identifier returned by `open` for reading or a network channel identifier returned by `open_connect` or `open_accept`.

If *re* is specified, it specifies a single character that marks the end of the record. In this case, `read` copies characters into *buf* until the *re* character is found and transferred into *buf*, or *len* bytes are read. If *re* is null character (\000) or omitted, then the data stream is not searched for a record end character.

Note

`read(fid, bu, 3000, "\n")` is equivalent to `getline(fid, buf)`.

When reading from a channel in non-blocking mode (from a callback established by `channel_set_callback`), the `read` may return fewer than *len* bytes. The return value is the actual number of bytes read. If the `read` would block, that is, there is no data pending on the channel, `read` returns `-2`. `read` also returns `-2` if *re* is given, but the data pending on the channel does not contain the record end character. In this case, Arbortext Editor queues the data internally until a complete record is available to return to a subsequent `read` from the channel callback read notification.

Unlike `getline`, `read` can handle binary data, that is, null characters. The following example is a function that can be used to make an exact copy of a file that may contain binary data. Note, that the “b” flag must be specified when opening both files.

```
function file_copy(from, to) {
local inf, outf
 inf = open(from, "rb")
 if (inf < 0) {
 response("Couldn't open file for read:" . from)
 return 0
}
 outf = open(to, "wb")
 if (outf < 0) {
 close(inf)
 response("Couldn't open file for write:" . to)
 return 0
}
 local buf, n=0, len
 while ((len = read(inf, buf, 512)) > 0) {
 n += write(outf, buf, len)
}
 close(outf)
 close(inf)
 return n;
}
```

read_preferences

`read_preferences` (*`path`*`[,` *`doc`*`]`)

This function reads the preferences stored in the preferences file specified by *path*, and uses them to update the current local scope (window, document, and view) values, updating the current window display. Note that the path to the preferences file cannot contain just single backslashes ("\"). The path must contain single forward slashes ("/"), double backslashes ("\\"), or single backslashes with the entire path enclosed in single quotes.

If the function executes successfully, it returns a one (`1`). If the *path* is invalid, the function returns a zero (`0`).

Note

The *doc* parameter is not currently used and may be removed in a future release.

Example:

```
read_preferences("C:/temp/arbortext.wcf")
```

```
read_preferences("C:\\temp\\arbortext.wcf")
```

registerApplicabilitySyntax

registerApplicabilitySyntax(*name*, *test_xpath*, *expr_path*, *parser_class*, *parser_namespace*[, *"parser_and_operator"*[, *"parser_or_operator"*[, *"parser_not_operator"*]]])

Registers a custom syntax for applicability.

- *name* — the name of the syntax
- *test_xpath* — an XPath expression that, when evaluated on an element, will advise if there is applicability for that element
- *expr_path* — an XPath expression that retrieves the applicability expression for a particular element
- *parser_class* — the classname of the parser for the syntax
- *parser_namespace* — the namespace used for the syntax

  The full namespaced name must be given so that the correct attribute can be added to the XML.

  This parameter is required to support editing of applicability settings. It is not used when publishing.
- *parser_and_operator* (optional)— the And operator used for syntax expressions in applicability attributes (default "&&")
- *parser_or_operator* (optional)— the Or operator used for syntax expressions in applicability attributes (default "||")
- *parser_not_operator* (optional) — the Not operator used for syntax expressions in applicability attributes (default "!!")

  The operator parameters are only required if the custom syntax requires operators that are different to the default ones.

For example, this syntax is registered for ATO, the default applicability type shipped with Arbortext Editor:

```
applic::registerApplicabilitySyntax("ATO", "boolean(@*[namespace
-uri() = 'http://arbortext.ptc.com/namespace/ATO' and local
-name() = 'applic'])", "@*[namespace-uri() = 'http://arbortext
.ptc.com/namespace/ATO' and local-name() = 'applic']" , "com
.ptc.arbortext.applicability.representation.ATOParser", "appl")
```

Arbortext Editor ships with two applicability syntaxes: APEX and ATO.

Once registered, a custom syntax must be assigned to the current environment using `inlineapplicabilitysyntax` (as set command or advanced preference) before it can be referenced.

remove_hook

remove_hook (*hookname*, *funcname*)

This function removes a callback function from the list of functions for the specified hook, where *hookname* is the name of the hook `set` option. The argument *funcname* is a string specifying the name of the user-defined function to call.

Refer to Hook Functions Introduction on page 115 for a detailed introduction to ACL hooks.

rename_ms_parameter

rename_ms_parameter (*entity*)

This function allows you to rename a marked section parameter entity.

entity is the name of an existing marked section parameter entity.

replace

replace(*searchStr*[, *replaceStr*[, *flags*[, *element*[, *doc*]]]])

This function searches for *searchStr* and replaces it with *replaceStr*. Returns `0` on failure. For success, if replace all is specified, then the number of replacements, else `1`.

- *searchStr* — The string to search for.
- *replaceStr* — The string to replace *searchStr*.
- *flags* — The bit mask for modifying the search.
  - `0x0001` — backward search (*-b*)
  - `0x0002` — case sensitive search (*-c*)
  - `0x0004` — *searchStr* is regular expression (*-e*)
  - `0x0008` — whole word only search
  - `0x0010` — *searchStr* contains markup (*-markup*)
  - `0x0020` — quiet search (*-q*) default is (*-noq*)
  - `0x0040` — don't prompt for wrapping at end of instance (*-wrapprompt*)
  - `0x0080` — wrap search (*-wrapscan*)
  - `0x0100` — don't wrap search (*-nowrapscan*) default is `-wrapscan` preference
  - `0x0200` — select *searchStr* when found (*-select*)

- `0x0400` — don't select *searchStr* when found (*-noselect*) default is `-selectscan` preference
- `0x0800` — search contents of entities (*-entityscan*)
- `0x1000` — don't search contents of entities (*-noentityscan*) default is `-entityscan` preference
- `0x2000` — replace all occurrences within instance
- `0x4000` — replace next occurrence of *searchStr*, then locate but don't replace the following occurrence (*-find*). Ignored if `0x2000` flag is set.

- *element* — Search only the contents of elements with this name
- *doc* — Current doc instance if not specified.

require (Function)

`require``(`*`package`*`[, `*`filename`*`])`

This function loads the package specified by *package* from the file *filename* if not already loaded. If *filename* is not specified, the list of directories specified in the `set loadpath` command option is searched for a file named *`package`*`.acl`, where `.acl` is the Arbortext Command Language file name suffix. Returns 1 (True) if the package was loaded or 0 (False) if the package file could not be opened or if the specified package is not defined after the file is read.

response

`response` `(`*`s1`*`[, `*`s2`*`,...])`

This function displays the message specified by *s1* in a panel with push buttons labeled by the following strings (*s2*, *s3*, and so on). This function returns the ordinal number of the button pressed by the user. If the dialog box is dismissed without a button press (by a window manager function, for example), this function returns `0`. However, if button two or three is **Cancel**, the function returns the ordinal number of the **Cancel** button. The prompt message may contain newline characters. If no button arguments are given, then an **OK** button is supplied. In this case, the `response` function displays the message and does not execute the next command until the user presses the **OK** button.

If the response function panel has just one push button, or if any button is labeled **OK** or **Yes**, the **ENTER** key works as an accelerator to activate the selection and dismiss the panel. The **ESCAPE** key is an accelerator for the cancel action if the `response` function panel has a button labeled **Cancel** or **No**.

To establish a mnemonic (hot key) for a particular dialog box item, precede the mnemonic letter with the '&' character in the associated string. Use “&&” anyplace you want a single '&' to show.

Example

```
response("Choose direction", "&Forward", "&Backward", "&Stop"
```

reverse

reverse(*s*)

This function returns a copy of string *s* with the characters in reverse order. For example, `reverse("abcd")` returns "dcba".

rindex

rindex(*s1*, *s2*)

This function returns the position in the string *s1* of the last (rightmost) occurrence of string *s2*, based at `1`. If the substring is not found, `0` is returned. If *s2* is the null string, this function returns `length(s1)`.

save_as_html_file

save_as_html_file ([*pathname*[, *overwrite*[, *doc*]]])

This function converts the specified document to an HTML file and saves it using the name specified by *pathname*. The presentation of the HTML file is based on the characteristics of the current Arbortext Editor stylesheet with tags off.

- *pathname* — Optional. Specifies full path for the resulting HTML file. If not supplied, the user will be prompted for the full path.
- *overwrite* — Optional. Specifies whether to automatically replace an existing file specified by *pathname* . If set to `0`, the user will be prompted to verify overwriting the file. If set to `1`, the file will be silently overwritten. The default value is `0`.
- *doc* — Specifies the document identifier of the document to be saved as HTML. The *doc* argument must be a return value from a previous call to doc_open on page 325, window_doc on page 591, or oid_doc on page 436. If *doc* is omitted or `0`, the current document is used.

For example,

```
$d=doc_open("c:/temp/artdoc.sgm");
save_as_html_file("c:/temp/artdoc.htm", 1, $d);
```

Note

The **File ▸ Save as HTML** menu option uses `save_as_html_file` when saving documents.

save_as_windchill_template_source

`save_as_windchill_template_source``()`

This function is only available when you are connected to a PTC server through the PTC Server connection in Arbortext Editor. The function enables you to save the currently open DITA topic document to the PTC server where it will be used as the source for a Dynamic Document template. Note that the function should only be used for a DITA topic template document opened from the **New Document** dialog box. The intention is to retain the feature that automatically sets the topic ID for the DITA template document stored on the PTC server.

Entering `save_as_windchill_template_source()` at the Arbortext Editor command line opens the **Save As Properties** dialog box. Use the dialog box to save the template to the desired location on the PTC server.

save_some_docs

`save_some_docs` (`[`*`noprompt`*`]`)

This function loops through the list of modified documents and prompts the user whether to save each one. If noprompt is true (non-zero), it saves all changes without prompting. The `exit_editor` function calls this function if its second argument is not 2.

The function returns `1` on success, `0` if there was an error saving a document, or `-1` if the user selected **Cancel** on a prompt to save a document.

schema_validate

`schema_validate` (`[`*`doc`*`]`)

Validates a document against a schema the user selects from a dialog box. The dialog box offers schemas specific to the namespaces used in the document and, at most, one non-namespaced schema.

- *doc* — The document to validate against the schema. If not specified, the current document is used.

The function returns one of the following values:

- `0` — The validation against one or more schemas failed.
- `1` — *doc* is valid.
- `-1` — The user canceled the validation.

Error results are displayed in the **Completeness Check Log** window and linked to the area of the document causing each error. If no errors are encountered, a message is sent to the status bar. Refer to `ns_schema_validate_batch` on page 425 and `schema_validate_batch` on page 509 for similar functionality when developing Arbortext Publishing Engine applications.

schema_validate_batch

`schema_validate_batch``(`*`schema`*`[, `*`doc`*`[, `*`flag`*`]])`

Validates a document against a non-namespaced schema.

- *schema* — The path and file name of a non-namespaced schema definition (`XSD`) file.
- *doc* — The document to validate against *schema*. If not specified, the current document is used.
- *flag* — Enables you to control how the *doc* is validated against the *schema*. The following flags are available:
  - `1` — Check identity constraints.
  - `2` — Check entities.
  - `4` — Check the document structure, including content and data types.
  - `8` — Check attribute values.
  - `16` — Report errors.
  - `32` — Define how to report errors.

    If set, either a list of errors or a message saying that there are no errors is written to an event log document. The event log can be retrieved by calling the ACL function `get_schema_log_doc()`, which will return the document ID as an integer. Use the document ID to inspect the content of the document, write it to disk, display it, or carry out manipulations as with an open document.

    If not set, any schema errors are reported in the same way as errors reported by a completeness check. Errors are reported in a popup **Parser Messages** window. If there are no errors, no window is displayed.
  - `64` — Fall back to the schema associated with *doc*, if the specified *schema* is not available or does not contain element definitions that occur in *doc*.

By default, the function checks all of these areas and reports errors (not to the buffer). If you only want to validate one or more of these areas, set the appropriate flag or flags. Add the flag values to set multiple flags. For example, to just check identity constraints and the document structure, set *flag* to `5` (`1 + 4`).

The function returns one of the following values:

- `0` — The validation failed.
- `1` — *doc* is valid.

All `schema_validate_batch` results are shown in the event-log document. Refer to `schema_validate` on page 508 for similar functionality.

scroll_to_oid

`scroll_to_oid` (*`oid`*`[,` *`pos`*`[,` *`adjcaret`*`]])`

This function positions the window displaying the document containing *oid* so that the object given by *oid*, *pos* is at column 1 on the top line of the screen. If *adjcaret* is specified and non-zero, then the cursor position is also moved so that the insertion point is visible.

seek

`seek``(`*`fid`*`,` *`offset`*`[,` *`origin`*`])`

This function sets the position of the next input or output operation on the file identified by *fid*, which must be a return value from a previous call to `open`.

The new position is given by *offset*, which is the signed distance offset bytes from the beginning, the current position, or the end of the file, according to the value of *origin* which must be 0, 1, or 2. If *origin* is omitted, 0 is used.

`seek` returns `0` on success or `-1` on failure, for example, if an attempt is made to seek on a file ID associated with a non-seekable device.

`seek(fid, 0)` rewinds the file associated with *fid*.

selected

`selected` (`[`*`doc`*`[,` *`type`*`]])`

This function returns `1` (True) if there is a selection in the current window. Note that this is faster than testing `length($selection)`.

- *doc* — Specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.
- *type* — A qualifier defining a particular type of selection:
  - `1` — Return true if this is a text selection.
  - `2` — Return true if this is a table selection.
  - `3` — Return true if this is either a text or table selection.

selected_element

`$oid = selected_element([doc[, flags]])`

This function returns the oid for the containing element if the current selection is limited to an entire element. An entire element includes the start and end tags and any content that they encompass. By definition, the `oid_select` function on page 456 selects entire elements.

This function returns the value of the `oid_null` function if there is no selection, or if the selection doesn't encompass an entire element (for example, start tag and content, but not the end tag).

The optional *doc* parameter specifies the identifier for the document containing the selection. If omitted or `0`, the value for the `current_doc` function on page 255 is used.

The *flags* parameter is an optional bitmask specifying various options, and is constructed by ORing the flags from the following list:

- `0x1` — Ignore an optional leading and/or trailing blank before considering the selection.

selection_anchor

`selection_anchor (pos[, doc])`

This function returns the oid of the object containing the anchored end of the selection in the window containing the document specified by *doc*, or the current document if omitted. Also returns the character offset of the end of the selection from oid in the output variable *pos*.

The anchor point of the selection is the point at which the selection was started, for example, the same as `selection_end` if the region was highlighted from right to left.

selection_balanced

`selection_balanced ([doc])`

This function returns `1` (True) if the selection in the current window is balanced, meaning that a tag pair (start and end tag) have been selected.

> **Note**
>
> `selected` must be True before `selection_balanced` will return a meaningful result.

The *doc* argument specifies the identifier of the document tree to query. If omitted or 0, the current document is used.

selection_end

`selection_end` (*`pos`*`[,` *`doc`*`]`)

This function returns the OID of the object containing the end of the selection in the window containing the document specified by *doc*, or the current document if omitted. Also returns the character offset of the end of the selection from OID in the output variable *pos*.

selection_has_change_tracking

`ret =` **`selection_has_change_tracking`**`(`[*`doc`*`])`

This function returns `1` if change records are found in the selected region of *doc*. If *doc* is not specified, the value of `current_doc` is used.

> **Note**
>
> The function will only return meaningful information if the Arbortext Editor selection is a region of text. If the selection includes one or more elements of a table (a grid, a row, a column, a cell, or several of any of the above) the function will return 0.
>
> To determine whether the current selection in the document is a text or a table selection, use the ACL function `selected()` on page 510.

selection_markup

`selection_markup` (`[`*`doc`*`[,` *`flags`*`]])`

This function returns a string representing the selected region in the given document (if no selection exists in that document, its value is null). The string includes any XML/SGML codes (markup).

flags — A bitmask that controls the markup included in the returned string and is constructed using OR with the following flags:

- `0x1` — Include the contents of the selected region. This is the default option if *flags* is `0` or unspecified.
- `0x2` — Include the XML or SGML header associated with the selected region. If the region does not include the entire document, this will be a fragment (file entity) header.
- `0x4` — Ignore the selection and act on the entire document instead.
- `0x8` — Use XML syntax in the string returned even if the selection is in an SGML document.
- `0x10` — Use SGML syntax in the string returned even if the selection is in an XML document.
- `0x20` — Don't convert non-ASCII characters into entity references (as though the `set writenonasciichar=char` option was in effect). If this bit is not specified, non-ASCII characters are converted according to the current `set writenonasciichar` on page 933 setting.
- `0x40` — Don't include insignificant line breaks. If this bit is not specified, insignificant line breaks will be included based on the `set outputrecordlength` command on page 858 setting.
- `0x80` — Suppress Arbortext processing instructions. Arbortext processing instructions can also be suppressed using the `set writepi` on page 935 command.
- `0x100` — Don't include the markup in the result. This option takes precedence over any other flag bits that control how markup is generated.

Calling `selection_markup` with no parameters will return the same string as the predefined variable *$selection* if `set sgmlselection=on` on page 885.

selection_start

`selection_start` (*`pos`*`[, `*`doc`*`]`)

This function returns the start of the selection in the window containing the document specified by *doc*, or the current document if omitted. Also returns the character offset of the start of the selection from OID in the output variable *pos*.

Examples

The functions `selection_start` and `selection_end` can be used to restore a selection, for example:

```
local o1, p1, o2, p2
o1 = selection_start(p1)
o2 = selection_end(p2)
 ...
goto_oid(o1, p1)
mark begin
goto_oid(o2, p2)
 mark end
```

set_custom_property

`set_custom_property` (*`key`*, *`value`*)

This function defines a value for the specified key that will be stored in memory. These keys and values can be used by customizations, applications, and repository adapters for configuration purposes.

- *key* — String value as defined by the customization, application or adapter.
- *value* — Any value, including an empty string.

`set_custom_property` returns any previous value associated with the specified key. This function will return an empty string if there is either no previous value or the previous value is an empty string.

Use the `get_custom_property` function on page 367 to retrieve a parameter and its associated value.

set_profile_group

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

Returns the profile configuration file markup for the specified set profile group as a string. If such a string cannot be constructed, an empty string will be returned.

- *set_profile_group_name* — The configuration file from which the markup is to be returned.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

set_profile_groups

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

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

- *arr* — Array of the names of the set profile groups specified in the profile configuration file.
- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

set_profile_groups_expressions

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.

- *arr* — Array of the names of the set profile groups and the portion of the markup from the profile configuration file that represents the profile classes or the relationships between profile values for the corresponding resolution group.

  For example, if *MyPublishProfile* is the name of the set profile group specified in the profile configuration file, the portion for:

  ```
  arr[MyPublishProfile]
  ```

  could be:

  ```
  <LogicalExpression>
   <LogicalGroup operator="OR">
   <LogicalGroup operator="AND">
   <LogicalNOT>
   <ProfileRef alias="Vehicle" value="Coupe">
   </LogicalNOT>
   <ProfileRef alias="Vehicle" value="Sedan">
   </LogicalGroup>
   <ProfileRef alias="Vehicle" value="Pickup">
   <ProfileRef alias="Vehicle" value="Sport Utility Vehicle">
   </LogicalGroup>
  </LogicalExpression>
  ```

- *doc* — Optional. The identifier of the document tree. If *doc* is omitted or is `0`, the current document is used.

set_user_property

set_user_property (*key*, *value*)

The `set_user_property` function defines a value for a specified parameter or variable that will be stored in memory. The parameter and its value are subsequently saved to the `arbortext.wcf` preferences file when the Arbortext Editor session exits. When Arbortext Editor exits, all parameters specified by `set_user_property` are saved to the user's preferences file. The next time

Arbortext Editor starts, these parameters and their values will be loaded into memory. A custom application can call this function anytime during a Arbortext Editor session.

To avoid conflicts with other applications that may be using the same memory during a Arbortext Editor session, use a unique qualified name (following the Java class naming convention) for the *key* parameter. For example, the *key* name could be something like `com.arbortext.sample.Verbose`, where `com.arbortext.sample` is the Java class for the application, and `Verbose` is the specific property.

The *value* parameter can be any value, including an empty string.

The function returns any previous value associated with the specified *key*. This function will return an empty string if there is either no previous value or the previous value is an empty string.

Use the `get_user_property` function on page 369 to retrieve a parameter and its associated value.

When Arbortext Editor starts, it loads these values into memory before processing the `custom` and `application` directories. This action makes the parameters available during the initialization phase of a custom application.

sgml_feature

`sgml_feature` (*`feature`*`[,` *`doc`*`]`)

This function returns information about one SGML Declaration feature for the document type associated with *doc*, or the current document if omitted. *feature* is one of the strings `subdoc`, `formal`, `concur`, `explicit`, `implicit`, `simple`, `rank`, `datatag`, `omittag`, or `shorttag`. The return value is `1`, `0`, or `-1`. Some features may be `1` (on) or `0` (off) and others are `0` or a positive number. This command returns `-1` if *feature* is invalid. The case of the string is not important.

show_composer_log

`show_composer_log` ()

The `show_composer_log` function displays the Event Log window. If there is no Event Log window, this function creates it.

smart_insert_categories

`smart_insert_categories`(*`arr`*`[,` *`doc`*`]`)

The `smart_insert_categories` function fills the array *arr* with the list of Smart Insert category names defined in the document type configuration file (`.dcf`) for the document type associated with *doc*. These categories are used in the **Insert Markup** dialog box when Smart Insert is turned on. If *doc* is omitted or `0`, the current document is used.

The function returns an array containing the category names in alphabetical order. The function also returns the number of entries in each array. The first entry is stored at index 1.

smart_insert_category_elements

`smart_insert_category_elements``(`*`arr`*`, `*`category`*`[, `*`doc`*`])`

The `smart_insert_category_elements` function fills the array *arr* with a list of element names for the Smart Insert *category* defined in the document type configuration file (`.dcf`) for the document type associated with *doc*. These category elements are used in the **Insert Markup** dialog box when Smart Insert is turned on. If *doc* is omitted or `0`, the current document is used.

The function returns an array containing the element names in the order in which they are declared in the `.dcf` file. The function also returns the number of entries in the array. The first entry is stored at index 1.

span

`span``(`*`s1`*`, `*`s2`*`)`

This function is like the C function `strspn` and returns the length of the initial substring of the string *s1* containing only characters in the string *s2*. Returns `0` if no characters in *s2* match.

spellskip_tag

`spellskip_tag` `(`*`tagname`*`[, `*`doc`*`])`

This function returns a `1` (true) if the element name *tagname* is listed in the current document type's `.dcf` file as an element whose content should be skipped when doing a spelling check.

The *tagname* parameter can be either a real element name as defined in the document type definition (DTD) or a tag alias configured by a user.

The optional *doc* parameter specifies the document identifier for the desired document instance and defaults to the return value for `current_doc` if omitted.

If the specified *tagname* is either an element whose contents should be spell checked, or an element that does not exist in the DTD, the function returns a `0` (false).

Examples

```
#Test computeroutput tag for spellcheck tag skipping.
#
$ret=spellskip_tag("computeroutput")
#Do something if the tagname in $elementname is
#set to be skipped.
if (spellskip_tag($elementname)) {
 ...
 )
```

split (Function)

split(*string*, *arr*[, *fs*])

This function splits the string represented by *string* into array *arr* and returns the number of fields. The string value of the optional third argument *fs* specifies the field separator. *fs* is a list of break characters. For example: `split("a:b;c", $arr, ":;")` splits the string `a:b;c` into three fields, storing `a` in `$arr[1]`, `b` in `$arr[2]`, and `c` in `$arr[3]`.)

The *fs* argument is optional; if omitted or null, the break characters are space, tab, and newline. Fields may be separated by multiple occurrences of white space. If *fs* is given, only a single instance of the field separator character separates each field. Any previous elements in the array `arr` are discarded before `split` stores the fields.

strcoll

`ret =` **strcoll** (*string_a*, *string_b*)

This function compares *string_a* with *string_b*according to the collation rules specified by the indexing collation configuration file indicated by the environment variable *APTIXLANG*.

The function returns `-1` if *string_a* collates before *string_b*, `0` if *string_a* and *string_b* sort the same, and `1` if *string_a* collates after *string_b*.

This function is useful in the `ixsorthook` function or if you wish to write your own index processing.

styler

styler([*oid*])

This function opens a new Arbortext Styler window or brings an existing one to the front for editing the current Editor view stylesheet for the document to which *oid* belongs. The oid element is selected in the Arbortext Styler **Elements** list. If no oid is specified, the default is `oid_current_tag(current_doc())`.

Note

If you are using the free version of Arbortext Styler to edit your Editor view stylesheet and have disabled Arbortext Styler in the `.dcf` file, then this function will not execute and generates an error message.

styler_enabled

styler_enabled ([*doc*])

This function returns a non-zero value if the stylesheet associated with *doc* can be modified using Arbortext Styler. This non-zero value indicates the Arbortext Styler version that the stylesheet was created by, or has been updated for use by. Updating from one Arbortext Styler version to the next happens automatically. If the stylesheet cannot be modified using Arbortext Styler, the function returns `0`.

This function uses the stylesheet for the current document if no document is specified.

styler_get_styled_elements

style_get_styled_elements(*sspath*[, *flags*])

This function takes a the stylesheet path *sspath* and returns the document identifier of a Free-form XML document containing a list of all of the styled elements in the stylesheet, that is, all of the elements with an assigned style other than **Unstyled**. If the stylesheet is not already loaded in memory, the function loads the stylesheet and leaves it loaded.

The *sspath* argument must be the path to a Arbortext Styler stylesheet.

The optional *flags* argument can have one of the following values:

- `0` — User Formatting Elements and Styler Formatting Elements are not included in the output. This is the default.
- `1` — User Formatting Elements are included in the output.
- `2` — Styler Formatting Elements are included in the output.
- `3` — User Formatting Elements and Styler Formatting Elements are both included in the output.

The Free-form document identified by this function has the following format:

```
<?xml version="1.0"?>
<doc namespace declarations>
<chapter/>
<section/>
```

```
<namespace:element/>
...
</doc>
```

The `doc` element is the wrapper element and contains all of the necessary namespace declarations used with elements in the stylesheet. Other elements represent the styled elements in the stylesheet. These elements are listed in alphabetical order in singleton syntax with any necessary namespace prefixes.

stylesheet

stylesheet ([*use*[, *doc*]])

This function returns the full path and file name for the stylesheet specified by *use* for a given document *doc*.

Valid values for *use* are:

Valid values for the *use* parameter

Value	Publishing Type
0	Editor
1	Print and PDF
2	HTML File
3	HTML Help
4	Web

If no document is specified, the *doc* parameter is the value of `current_doc`.

stylesheet_export_fosi

stylesheet_export_fosi([*path*[, *doc*]])

This function writes the current editor Arbortext Styler stylesheet (`.style`) to *path*. If *path* is not specified, the **Export FOSI Stylesheet** dialog box is launched to let the user specify the path. When no *doc* is specified, Arbortext Editor defaults to the document returned by `current_doc`. The function returns `1` if the stylesheet is successfully exported, and `0` if there is any error. An error is returned if the current editor stylesheet can not be modified by Arbortext Styler.

stylesheet_export_xsl

stylesheet_export_xsl([*path*[, *doc*[, *target*]]])

This function exports the current editor Arbortext Styler stylesheet (`.style`) for *doc* as an XSL stylesheet to the path specified by *path*. The `stylesheet_export_xsl` function is only valid when the stylesheet for *doc* is a Arbortext Styler stylesheet. If *doc* is not supplied, it defaults to `current_doc`. If *path* is not specified, a dialog box is launched to let the user specify the path.

The *target* parameter must be on the following types:

- `fo` — Exports an XSL-FO stylesheet for creating print or PDF output.
- `html` — Exports an XSL stylesheet for creating HTML output.
- `htmlhelp` — Exports an XSL stylesheet for creating HTML Help output using the Arbortext Editor **Publish ► For HTML Help** capability.
- `web` — Exports an XSL stylesheet for creating web output using the Arbortext Editor **Publish ► For Web** capability.

If *target* is not specified, `html` is assumed.

The function returns 1 if the stylesheet was successfully exported, 0 if there was an error.

stylesheet_gentext_lang_stats

`stylesheet_gentext_lang_stats``(`*`lang`*`, `*`stats`*`[, `*`doc`*`])`

This function provides information about the translation for the specified *lang* imported from an XLIFF file into a stylesheet to provide translations of generated text. *stats* populates an array that provides information on the results of the requested translation from the XLIFF file. *doc* is optional - specify a stylesheet document or a user document with which the stylesheet is associated.

Values for the parameters are listed below:

Parameter values

Parameter	Permitted Values
lang	The language code of the language for which to obtain statistics
stats	The following values are assigned to the array: stats[‘current’] The number of translation units that are current stats[‘effectivecurrent’] The number of translations units in effective definitions that are current stats[‘notcurrent’] The number of translation units with a translation that is not current (the source has changed after the translation was added) stats [‘effectivenotcurrent’] The number of translation units in effective definitions with a translation that is not current stats[‘untranslated’] The number of translation units without a translation stats [‘effectiveuntranslated’] The number of translation units in effective definitions without a translation stats[‘total’] The total number of translation units in the stylesheet
doc	(Optional) Path to the stylesheet or user document. If not specified, the function will refer to the current document.

When successful, the function will return 1, otherwise it will return 0. The function can be unsuccessful if:

- Stylesheet is not found
- *lang* does not match a target generated text language on the XLIFF file

If the function is unsuccessful, all *stats* will be zero.

Refer to *Using ACL to Import XLIFF Files* in Arbortext Styler help for some sample ACL code demonstrating the use of this function.

stylesheet_get_list_dir

stylesheet_get_list_dir(*labels[]*, *paths[]*, *is_Styler[]*, *use*[, *dir*[, *maxlabelchars*[, *labeltitleonly*]]])

This function returns information on the valid stylesheets in directory *dir* in the *labels*, *paths*, and *is_Styler* arrays.

The labels for valid stylesheets in directory *dir* fill the *labels[]* array. Arbortext Editor generates labels by combining the stylesheet title (if any) and path into a string and truncating it if necessary so that it is no longer than *maxlabelchars*. A stylesheet title is specified in the Stylesheet ID processing instruction in the stylesheet file (<?APT StylesheetID ... Title="title" ...>). These labels are displayed in the Arbortext Editor user interface.

The absolute path of each valid stylesheet fills the *paths[]* array.

Values of `0` and `1` fill the *is_Styler[]* array. A `1` indicates you can use the stylesheet with Arbortext Styler, a `0` indicates that you cannot use the stylesheet with Arbortext Styler.

Valid stylesheets are `.style`, `.xsl`, and `.fos` files whose publishing type matches *use*. A stylesheet's type is derived from the **CompositionTypes** attribute on the **StylesheetID** processing instruction. Valid *use* types are `any`, `htmlfile`, `htmlhelp`, `print`, `pdf`, `web`, `wordxml`, and `xsl`. If *use* is `any`, all `.style`, `.xsl`, and `.fos` files are returned.

The following table describes the default stylesheet publishing types if a stylesheet file does not have a StylesheetID processing instruction, and whether editor will be added:

Stylesheet type	Default publishing types	Valid screen stylesheet	Can edit using Arbortext Styler
FOSI (`.fos` file)	print, pdf, htmlfile	Yes	No
XSL (`.xsl`)	print, pdf, htmlfile, xsl	Yes	Yes
XSL-FO	print, pdf, xsl	No	No
Other XSL	xsl	No	No
Arbortext Styler (`.style`)	print, pdf, htmlfile, htmlhelp, web	Yes	Yes

If *dir* is omitted, the current directory is used.

maxlabelchars controls the maximum number of characters included in the returned labels. If *maxlabelchars* is omitted, the label length defaults to 50.

If *labeltitleonly* is `1`, then the label is the stylesheet title, if it exists. If the stylesheet title doesn't exist, the label is the path to the stylesheet.

stylesheet_get_list_doc

`stylesheet_get_list_doc``(`*`labels[]`*`, `*`paths[]`*`, `*`is_styler[]`*`, `*`use`*`[, `*`doc`*`[, `*`maxlabelchars`*`[, `*`labeltitleonly`*`]]])`

This function returns information on valid stylesheets for *doc* in the *labels*, *paths*, and *is_styler* arrays.

The labels for valid *doc* stylesheets fill the *labels[]* array. Arbortext Editor generates labels by combining the stylesheet title (if any) and path into a string and truncating it if necessary so that it is no longer than *maxlabelchars*. A stylesheet title is specified in the Stylesheet ID processing instruction in the stylesheet file (<?APT StylesheetID ... Title="title" ...>). These labels are what Arbortext Editor displays in the user interface.

The absolute path of each valid stylesheet fills the *paths[]* array.

Values of `0` and `1` fill the *is_styler[]* array. A `1` means you can use the stylesheet with Arbortext Styler, a `0` means that you cannot use the stylesheet with Arbortext Styler.

Valid stylesheets are `.style`, `.xsl`, and `.fos` files whose publishing type matches *use*. A stylesheet's type is derived from the **CompositionTypes** attribute on the **StylesheetID** processing instruction.

Valid *use* types are `any`, `htmlfile`, `htmlhelp`, `print`, `pdf`, `web`, `wordxml`, and `xsl`. If *use* is `any`, all `.style`, `.xsl`, and `.fos` files are returned.

The following table describes the default stylesheet publishing types if a stylesheet file does not have a StylesheetID processing instruction:

Stylesheet type	Default publishing types	Valid screen stylesheet	Can edit using Arbortext Styler
FOSI (`.fos` file)	print, pdf, htmlfile	Yes	No
XSL (`.xsl`)	print, pdf, htmlfile, xsl	Yes	Yes
XSL-FO	print, pdf, xsl	No	No
Other XSL	xsl	No	No
Arbortext Styler (`.style`)	print, pdf, htmlfile, htmlhelp, web	Yes	Yes

If *doc* is omitted or `0`, the current document is used.

maxlabelchars controls the maximum number of characters included in the returned labels. If *maxlabelchars* is omitted, the label length defaults to 50.

If *labeltitleonly* is `1`, then the label is the stylesheet title, if it exists. If the stylesheet title doesn't exist, the label is the path to the stylesheet.

stylesheet_import_xlf

`stylesheet_import_xlf``(`*`path`*`, `*`flags`*`, `*`stats`*`[, `*`doc`*`])`

This function imports the XLIFF file (`.xlf`) found at *path* into a stylesheet to provide translations of generated text. Use *flags* to control how to process translations when certain questions occur. *stats* populates an array that provides information on the number of translation units that were found in the XLIFF file and updated in the stylesheet, and the number that not imported or not found in the XLIFF file. *doc* is optional - specify a stylesheet document or a user document with which the stylesheet is associated.

Values for the parameters are listed below:

Parameter values

Parameter	Permitted Values
path	The path to a single XLIFF file
flags	Use a value of 0 or a combination of the following values: 1 — Mark translations as current when they are identical to the source 2 — Import translations that are already marked as current in the stylesheet 4 — Import translations when the source has changed but the target remains the same For example, use a value of `7` to match against all the criteria.

Parameter values (continued)

Parameter	Permitted Values
stats	The following values are assigned to the array: stats[‘updated’] — Number of translation units updated during the import stats[‘rejected’] — Number of translation units not imported for one of the 3 reasons described in *flags* stats[‘missing’] — Number of translation units in the stylesheet but without equivalent in the XLIFF file stats[‘found’] — Number of translation units in the stylesheet with equivalent in the XLIFF file stats[‘xlftotal’] — Total number of translation units in the XLIFF file stats[‘sstotal’] — Total number of translation units in the stylesheet
doc	(Optional) Path to the stylesheet or user document. If not specified, the function will refer to the current document.

When successful, the function will return 1, otherwise it will return 0. The function can be unsuccessful if:

- Stylesheet is not found
- XLIFF file cannot be opened
- XLIFF file is not valid
- One of more stylesheet modules is read only, preventing a translation being imported

If the function is unsuccessful because of the read-only nature of a stylesheet module, generated text in the writable modules will still be updated and the results reflected in the import statistics. If it is unsuccessful for any of the other listed reasons, all *stats* will be zero.

Refer to *Using ACL to Import XLIFF Files* in Arbortext Styler help for sample ACL demonstrating the use of this function.

stylesheet_list_add

stylesheet_list_add (*path*, *use*, *doctype*)

This function adds *path* to a list of stylesheet paths for the specified *doctype* and *use*. The stylesheet_get_list_doc function on page 524 searches this list for valid stylesheets during the current Arbortext Editor session. Arbortext Editor verifies that the *path* specifies a stylesheet that supports the *use*.

Valid *use* types are `any`, `htmlfile`, `htmlhelp`, `pda`, `print`, `pdf`, `web`, `wml`, and `xsl`. If *use* is `any`, all `.style`, `.xsl`, and `.fos` files are returned.

For a stylesheet to be returned by `stylesheet_get_list_doc`, the *use* passed to `stylesheet_get_list_doc` must match the *use* passed to `stylesheet_list_add`.

The *doc* passed to `stylesheet_get_list_doc` must use the same *doctype* passed to `stylesheet_list_add`.

If you are using Arbortext Publishing Engine for publishing documents, the *path* must be specified by a URL or it will be ignored by Arbortext Editor. A stylesheet used for editing must always be specified using a local absolute or relative path.

An example for adding stylesheets using `stylesheet_list_add`:

```
# Let users select a stylesheet for a given "doc" and "use"
# from a list of stylesheets in their document and document
# type directories or by browsing; if they browse to select
# a stylesheet, include that stylesheet in the list the
# next time they ask for the same "use" with either the
# same document, or another document of the same document
# type.
# Valid "use" values are any of the Type attributes
# of publish elements in the document type's .dcf file,
# plus "editor"
function stylesheet_get(doc, use)
 {
 # First get list of stylesheets that we show by label;
 #
 # The list includes those valid for "use" in the
 # document directory, document type directory, or
 # that were previously selected for this use and
 # document type;
 local labels[], paths[], is_styler[];
```

```
 local n = stylesheet_get_list_doc(labels, paths, is_styler, use, doc);
 # Display this list to users, let them pick a stylesheet
 # from the list or by browsing.
 # NOTE: this function does not actually exist;
 local path = stylesheet_pick(labels, paths, use);
 # If users browsed to the stylesheet, add it to the
 # list of stylesheets that stylesheets_get_list_doc
 # will consider returning the next time;
 stylesheet_list_add(path, use, doc_type(doc));
 return path;
}
```

stylesheet_new

stylesheet_new ([*doc*])

This function creates a new stylesheet using an existing template, makes this new stylesheet the Editor view stylesheet, and launches Arbortext Styler.

The temporary name `Stylesheet`*n* is assigned to the new stylesheet; it displays on the document title bar. The number *n* increments during an entire Arbortext Editor session.

Note

If you are using the free version of Arbortext Styler to edit your Editor view stylesheet and have disabled Arbortext Styler in the `.dcf` file, then this function will not execute and generates an error message.

stylesheet_revert

stylesheet_revert ([*doc*])

This function causes the editor stylesheet for document *doc* to revert to the last saved version. If *doc* is not supplied, the value of `current_doc` is used. This function is enabled only if the current stylesheet has been used before. The function is disabled when working with a new stylesheet.

This function corresponds to the **Format ▸ Revert Stylesheet** menu item.

stylesheet_save

stylesheet_save ([*doc*])

This function saves the current editor stylesheet for document *doc*; if *doc* is not supplied, the function defaults to `current_doc`. Arbortext Styler stylesheets are saved as `.style` files. FOSIs are saved as `.fos` files. If the stylesheet has not previously been saved, this function launches the **Save Stylesheet As dialog** box.

This function corresponds to **Format ► Save Stylesheet**.

stylesheet_saveas

`stylesheet_saveas` `([`*`pathname`*`[, `*`doc`*`]])`

This function lets you save a modified stylesheet. If the current stylesheet was created or modified by Arbortext Styler, it is automatically saved as a `.style` file.

This function will save the stylesheet in one of three ways:

- As a document level stylesheet, using the current *`doc`* directory as the target directory, and the current *`docname`* to create a document-specific stylesheet, (the resulting stylesheet is written to *`doc-pathname/docname`*`.xsl`)
- As an application level stylesheet, using the current *`doc`* directory as the target directory, and the current *`application-name`* to create a local version of the application stylesheet (the resulting stylesheet is written to *`doc-pathname/application-name`*`.xsl`)
- As an independent stylesheet, using a target directory and name specified by the user.

If no *pathname* is supplied, this function launches the **Save Stylesheet As** dialog box for document *doc*.

If *doc* is omitted or `0`, the current document is used.

The two possible return values from the dialog box are:

OK	The function returns the path the stylesheet was saved to.
Cancel	The function returns an empty string.

The **Save Stylesheet As** menu item corresponds to `stylesheet_saveas`.

stylesheet_select

`stylesheet_select` `([`*`name`*`[, `*`use`*`[, `*`doc`*`]]])`

This function controls the selection of a stylesheet; it corresponds to the **Format ► Select Stylesheets** menu item. All parameters are optional.

If you do not supply a *name*, the function launches the **Select Stylesheets** dialog box for document *doc*; (the value of `current_doc()` is used if you do not specify *doc*).

If you supply a *name*, this function selects the named stylesheet and returns its full path name. *name* can be specified in the following ways:

- The name can be the path and file name of the stylesheet, with or without the `.fos`, `.xsl`, or `.style` extension. Arbortext Editor first looks for a `.fos` file, and then an `.xsl` file, and then a `.style` file.
- The name can be the base name of the stylesheet file (the file name without the path). If just the base name is supplied, Arbortext Editor looks for the file in the document directory, and, if not found, in the application directory.

  The *use* parameter determines what use the stylesheet is selected for:

  - If `use` = `0`, the function selects the stylesheet for the Editor view.
  - If `use` = `1`, it selects the stylesheet for print published output (printing and print preview, publish PDF file).
  - If `use` = `2`, it selects the stylesheet to be used for **Save as HTML** or **Publish ► HTML File**.
  - If `use` = `3`, it selects the stylesheet to be used for or **Publish ► For HTML Help**.
  - If `use` = `4`, it selects the stylesheet to be used for or **Publish ► For Web**.

sub

sub(*regexpr*, *repl*[, *string*, *lvalue*])

The `sub` function is similar to `gsub`, except that this function attempts only one match and substitution of *string* that matches the regular expression *regexpr* and replaces the substring with the string *repl*. The resulting string is assigned to *lvalue*, which must be a variable name or an array element. If *string* is omitted, then the value of *lvalue* is used as the source string. When copying the *string* to *lvalue*, the part that was matched by *regexpr* is replaced according to the string *repl*.

If *repl* contains a `&` or `\0`, then it is replaced with the substring that matched *regexpr*. If *repl* contains a `\` *n*, where *n* is a digit between 1 and 9, then it is replaced in the substitution with that part of the *string* that matched the *n* th parenthesized subexpression of *regexpr*. *n* must not be greater than the number of parenthesized expressions in *regexpr*.

`sub` returns `1` (True) if the substring was replaced, or `0` if no match occurred. The *lvalue* is not changed if `0` is returned. Note that because the expression parser interprets backslash sequences in double quoted strings, you must use two backslashes to get a single `\` character in *regexp* or *repl* or use single quotes to delimit strings.

substr

substr(*string*, *n1*[, *n2*])

This function extracts a substring from *string*, starting at position *n1*, of *n2* characters. The first character of *string* is position 1. If *n1* is larger than `length(`*string*`)`, the null string is returned. If *n1*+*n2* is longer than *string*, then the remainder of *string* is returned. The *n2* argument is optional; if omitted, the remainder of the string is returned.

For example, the value of `substr("rnotes9302", 2, 5)` is `"notes"`.

system

system(*cmd*[, *flags*])

Use this function to run applications outside of the Arbortext Editor environment. The *cmd* parameter is a command to execute in a DOS window. If *flags* is set to one (`1`) the command will be executed in a hidden window.

If the process fails to launch, the function returns a negative one (-1) and stores an error message in the *$main::ERROR* variable.

system_id

system_id ([*doc*])

This function returns the system identifier from the `DOCTYPE` declaration for the document *doc* or the null string if none was specified. If *doc* is omitted or `0`, the current document is used.

tag_alias

tag_alias (*tagname*[, *doc*])

This function returns the alias of *tagname*, where *tagname* is the real name of the tag as defined in the DTD. A null string is returned if *tagname* does not have an alias.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

tag_attr_choices

tag_attr_choices (*tagname*, *attr*, *arr*[, *doc*])

This function fills the array *arr* with the list of declared values for the NAMEGROUP type attribute named *attr* of the element specified by *tagname*, returning the number of choices. The first attribute value is returned as index 1.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or a real names. However, this function will return the real name of the attribute value, not its alias.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or 0, the current document is used.

Note

If the document type definition for your document instance includes the NAMECASE GENERAL NO setting, then the *tagname* parameter will be case sensitive.

tag_attr_conref

tag_attr_conref (*tagname*, *attr*[, *doc*])

This function returns 1 (True) if the attribute *attr* for the element specified by *tagname* is declared as a #CONREF attribute in the DTD for the specified document. If the document type definition for your document instance includes the NAMECASE GENERAL NO setting, then the *tagname* parameter will be case sensitive. (If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names.)

The *doc* argument specifies the identifier of the document tree. If omitted or 0, the current document is used.

Note

This function applies to SGML documents only. It is not for use with XML documents or DITA content references.

tag_attr_default

tag_attr_default (*tagname*, *attr*[, *doc*])

This function returns the default value of the attribute *attr* for the element *tagname* as defined in the DTD for the specified document.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names. However, this function will return the real name for the attribute value, not its alias.

It returns a null string if *attr* does not have a default value or if no such attribute or element exists. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_attr_fixed

tag_attr_fixed (*tagname*, *attr*[, *doc*])

This function returns 1 (True) if the attribute named *attr* for the element *tagname* is declared as a `#FIXED` attribute in the DTD for the specified document.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names.

If *doc* is omitted, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_attr_required

`tag_attr_required` (*`tagname`*, *`attr`*[, *`doc`*])

This function returns 1 (True) if the attribute named *attr* for the element named *tagname* is declared as a `#REQUIRED` attribute in the DTD for the specified document.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names.

If *doc* is omitted, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_attr_type

`tag_attr_type` (*`tagname`*, *`attr`*[, *`doc`*])

This function returns the declared attribute value type for the attribute named *attr* for the element specified by *tagname*. The returned value is one of the following strings:

"CDATA"

"ENTITY"

"ENTITIES"

"ID"

"IDREF"

"IDREFS"

"NAME"

"NAMES"

"NMTOKEN"

"NMTOKENS"

"NOTATION"

"NUMBER"

"NUMBERS"

"NUTOKEN"

"NUTOKENS"

"NAMEGROUP" (if the attribute was declared as a name group)

.

> **Note**
>
> If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

> **Note**
>
> If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_attr_value

`tag_attr_value` (*`tagname`*, *`attr`*[, *`doc`*])

This function returns the current global value of the attribute named *attr* for the tag specified by *tagname*.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names. However, this function will return the real name of the attribute value, not its alias.

If *attr* is not a valid attribute or *tagname* is not a valid tag, a null string is returned.

The *doc* argument specifies the identifier of the document tree. If *doc* omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_attrs

`tag_attrs` (*`tagname, arr[, doc]`*)

This function fills the array *arr* with the list of attribute names declared for the tag specified by *tagname*. This function returns the number of attributes. The first attribute name is returned as index 1.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name. However, this function will return the real name of an attribute, not its alias.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_content

`tag_content` (*`tagname`*`[, `*`doc`*`]`)

This function returns the declared content of the tag specified by *tagname*. The returned value is one of the following strings:

"CDATA"

"RCDATA"

"EMPTY"

"MIXED"

"ANY"

"ELEMENT"

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

If the tag is a Arbortext Editor defined tag (such as `_font`), the function returns "`UNDEFINED`". If the tag is not declared in the DTD and is not a Arbortext Editor defined tag, the function returns an empty string.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_create

tag_create (*qname*[, *uri*[, *empty*[, *doc*]]])

This function is used to either create new tags in Free-form XML document instances or to create foreign namespace tags in XML documents. The *qname* parameter provides the name of the new tag with an optional prefix. The *uri* parameter, if supplied, specifies the URI for the new tag. If it is not specified, then the target namespace for the document is used (none for a Free-form XML document). If the *empty* parameter is supplied and not zero, then the element is not allowed to have content.

The function returns `1` on success.

If the *doc* parameter is `0` or omitted, then the current document is used.

tag_description

tag_description (*tagname*[, *doc*])

This function returns the description for the specified tag. A null string is returned if *tagname* does not have an description. The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

tag_display (Function)

tag_display (*tagname*[, *doc*])

This function returns a string describing the global tag display mode of the tag specified by *tagname*.

> **Note**
>
> If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

If *doc* is omitted or `0`, the current document is used.

The returned string corresponds to the option set by the `tag_display` command and is one of the following: "full", "partial", "icon", "none", or "default". If the tag display mode has not been changed for *tagname*, the function returns "default".

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_display_name

`tag_display_name` (*`tagname`*`[, `*`doc`*`]`)

This function returns the alias of the tag specified by *tagname*, if the tag has an alias. If the tag doesn't have an alias, `tag_display_name` returns the tag's real name. *tagname* can be an alias or a real name.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is searched.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then *tagname* is case sensitive.

tag_exists

`tag_exists` (*`tagname`*`[, `*`doc`*`]`)

This function returns 1 (True) if the tag named *tagname* is a valid tag name (including user-defined tags) in the current document (regardless of whether the tag is currently in use). If the tag name is not valid, 0 is returned.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_has_attr

`tag_has_attr` (*`tagname`*`, `*`attr`*`[, `*`doc`*`[, `*`dtdOnly`*`]])`

This function returns 1 (True) if the tag *tagname* has an attribute named *attr*. If the tag doesn't have the specified attribute, 0 is returned.

Note

If you have applied an alias map to the document, *tagname* and *attr* can be either aliases or real names.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

If the value specified by *dtdOnly* is omitted or is `1`, then the function will only return `1` for tags and attributes that are declared in the DTD or XML schema. If the value specified by *dtdOnly* is `0`, then the function will also return `1` if *attr* is an undeclared attribute that has been seen on the tag *tagname*.

For a Free-form XML document, the *dtdOnly* parameter is ignored. The function will return `1` if *attr* is seen on the tag *tagname*.

Note

If the DTD for your document includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.

tag_has_conref

`tag_has_conref` (*`tagname`*`[, `*`doc`*`])`

This function returns 1 (True) if the tag specified by *tagname* has an attribute declared as `#CONREF` in the DTD. If the tag doesn't have this attribute, the function returns 0. If the DTD for your document includes the `NAMECASE`

GENERAL NO specification, then the *tagname* parameter will be case sensitive. (If you have applied an alias map to the document, *tagname* can be either an alias or a real name.)

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or 0, the current document is used.

Note

This function applies to SGML documents only. It is not for use with XML documents or DITA content references.

tag_names

tag_names (*arr*[, *dtd*[, *doc*]])

The tag_names function fills the array *arr* with an alphabetical list of elements and similar markup icons for the current document type, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

Note

Even if you have applied an alias map to the document, this function will return an element's real name, not its alias.

If the value specified by *dtd* is non-zero, then the array includes only names of elements defined in the DTD.

If *dtd* is omitted or is 0, the array *arr* includes:

- The DTD element names.
- The names of markup icons supplied by Arbortext Editor. (For example, _ignore, _font, and so on.)

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or 0, the current document is used.

tag_names_ns

tag_names_ns (*arr*[, *namespace*[, *doc*]])

The `tag_names_ns` function fills the array *arr* with an alphabetical list of tags for the given document. If the *namespace* parameter is specified and not empty, then only those tags declared with that namespace URI are returned. If the *namespace* parameter is specified and is an empty string, then only those tags with no namespace are returned. If the *namespace* parameter is not specified, then all declared tags in the document type are returned, as well as tags in any foreign namespaces with the names qualified in the form `{`*`namespace-uri`*`}`*`local-name`*.

The function returns the number of entries loaded in the array.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

This function is meant for XML documents. If *doc* refers to a non-XML document, the *namespace* parameter is ignored, and the function acts like `tag_names`(*arr*, 1, *doc*) was called.

tag_real_name

`tag_real_name` (*`tagname`*`[,` *`doc`*`]`)

This function returns the real name (as specified in the DTD) of *tagname*. The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is searched.

tag_substitutions

`tag_substitutions` (*`tagname`*`,` *`arr`*`[,` *`doc`*`]`)

This function fills the array *arr* with the list of elements specified in the `.dcf` file as being possible substitutions for the element specified by *tagname*. The function returns the number of elements added to the array. The order of the array matches the order specified in the `.dcf` file.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name. However, `tag_substitutions` will return the real names of the elements, not their aliases.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, the *tagname* parameter will be case sensitive.

target_id_attr_name

target_id_attr_name (*tagname*[, *doc*])

This function returns the name of the ID attribute (for example, `id`) for the target tag specified by *tagname*. If the tag has no such attribute, `target_id_attr_name` returns the null string.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name. However, this function will return an attribute's real name, not its alias.

The *doc* argument specifies the identifier of the document tree. If *doc* is omitted or `0`, the current document is used.

target_tag

target_tag (*tagname*[, *doc*])

The `target_tag` function returns `1` (true) if the tag named *tagname* is declared as a target element in the `.dcf` file for the document type associated with *doc*. If *doc* is omitted or `0`, the current document is used.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

target_tag_name

target_tag_name ([*doc*])

The `target_tag_name` function returns the name of the primary target element for the document type associated with *doc*. It returns the null string if no target tag was designated for the document type, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

Note

Even if you have applied an alias map to the document, this function will return an element's real name, not its alias.

tbl_area_celllist

`count = tbl_area_celllist(upperLeftCellId, lowerRightCellId, array)`

This function fills *array* with the ID of each cell in the rectangle of cells defined by *upperLeftCellId* and *lowerRightCellId* and returns the number of IDs. The IDs are returned in row major order.

tbl_caret_col

`tbl_caret_col ()`

This function returns the column number (starting with 1) of the cell containing the cursor. Returns 0 if the cursor is not in a table cell.

tbl_caret_row

`tbl_caret_row ()`

This function returns the row number (starting with 1) of the cell containing the cursor. Returns 0 if the cursor is not in a table cell.

tbl_cell_col

`columnId = tbl_cell_col(cellId)`

This function returns the ID of the column containing cell *cellId*.

tbl_cell_clear

`[0|1] = tbl_cell_clear(cellId, content, span, attributes)`

If the *content*, *span*, and *attribute* parameters are non-zero values, `tbl_cell_clear`deletes the content and attributes of *cellId*and deletes the multicell that contains *cellId*.

tbl_cell_fontpi

`oid` = **`tbl_cell_fontpi`**`(`*`cellId`*`, `*`command`*`)`

This function attempts to insert, locate, or delete a cell font processing instruction for cell *cellId* as indicated by *command*, which may have the value `insert`, `find`, or `delete`. After doing so, this function returns the OID of the cellfont PI, if it still exists, or `null_oid`.

tbl_cell_in_multicell

`[0|1]` = **`tbl_cell_in_multicell`**`(`*`cellId`*`)`

This function returns 1 if cell *cellId* is a member of a multicell, 0 otherwise. Note that this routine is equivalent to the expression `tbl_cell_is_spanning(`*`cellId`*`) || tbl_cell_is_spanned(`*`cellId`*`)`.

tbl_cell_instantiate

`[0|1]` = **`tbl_cell_instantiate`**`(`*`cellId`*`)`

This function marks cell *cellid* as being non-sparse. Some table models, such as CALS, allow sparse markup, where some cells of a table are not explicitly described in markup. We add “generated” cell markup to the document when we read a sparse table so that there is markup underlying every table cell. When we save a document containing a table, we delete this “generated” markup, unless the cell has acquired content, attributes, or some other reason for existence. This function allows the user to require that the markup corresponding to a cell NOT be discarded, even if it is generated markup.

Note that we never strip out “real” markup, only generated markup.

tbl_cell_is_spanned

`[0|1]` = **`tbl_cell_is_spanned`**`(`*`cellId`*`)`

This function returns 1 if cell *cellId* is a member of a multicell but not the master (spanning) cell, 0 otherwise.

tbl_cell_is_spanning

`[0|1]` = **`tbl_cell_is_spanning`**`(`*`cellId`*`)`

This function returns 1 if cell *cellId* is the master (spanning) cell of a multicell, 0 otherwise.

tbl_cell_neighbor

`cellId = **tbl_cell_neighbor**(cellId, direction, offset)`

This function returns the ID of the cell *offset* cells away in direction *direction* from cell *cellId*. It returns *h::tblNullToid* if the surrounding grid doesn't contain a cell as far away as specified.

tbl_cell_next_galley_cell

`cellId = **tbl_cell_next_galley_cell**(cellId[, wrap])`

This function returns the ID of the cell after cell *cellId* in galley order, or h: tblNullToid if no such cell exists. If *wrap* is present and non-zero, `tbl_cell_next_galley_cell` will return the ID of the first cell in the sequence when the last cell in the sequence is reached.

tbl_cell_on_multicell_edge

`[0|1] = **tbl_cell_on_multicell_edge**(cellId, direction)`

This function returns 1 if cell *cellId* is on a specific edge of a multicell, 0 otherwise.

tbl_cell_prev_galley_cell

`cellId = **tbl_cell_prev_galley_cell**(cellId[, wrap])`

This function returns the ID of the cell before cell *cellId* in galley order, or h: tblNullToid if no such cell exists. If *wrap* is present and non-zero, `tbl_cell_next_galley_cell` will return the ID of the first cell in the sequence when the last cell in the sequence is reached.

tbl_cell_row

`rowId = **tbl_cell_row** (cellId)`

This function returns the ID of the row containing cell *cellId*.

tbl_cell_ruleneighbor

`ruleId = **tbl_cell_ruleneighbor**(cellId, direction)`

This function returns the ID of the rule bounding cell *cellId* on side *direction*.

tbl_cell_setcaret

`[0|1] = tbl_cell_setcaret(cellId, end)`

This function places the caret at the beginning (if *end* is zero) or end of cell *cellId.*

tbl_cell_span

`[0|1] = tbl_cell_span(upperLeftCellId, lowerRightCellId)`

This function combines the rectangle of cells specified by *upperLeftCellId* and *lowerRightCellId* into a multicell.

It does this by constructing a new rectangle with the coordinates [sumUpperLeft, sumLowerRight]

where:

```
sumUpperLeft.X = min(oldUpperLeft.X, newUpperLeft.X)
sumUpperLeft.Y = min(oldUpperLeft.Y, newUpperLeft.Y)
sumLowerRight.X = max(oldLowerRight.X, newLowerRight.X)
sumLowerRigth.Y = max(oldLowerRight.Y, newLowerRight.Y)
```

and trying to span that.

This will fail if any cell in the resulting rectangle is a member of any other multicell.

tbl_cell_unspan

`[0|1] = tbl_cell_unspan(cellId)`

This function deletes the multicell containing *cellId*, which may be any member of the multicell.

tbl_cell_unspanned_neighbor

`cellId = tbl_cell_unspanned_neighbor(cellId, direction, offset)`

This function returns the ID of the cell *offset* cells away in direction *direction* from cell *cellId*; it considers only unspanned cells in calculating how far to go.

tbl_col_cell

`cellId = tbl_col_cell(columnId, rowIndex)`

Returns the ID of the cell that is positioned *rowIndex* rows from the top of column *columnId*. The uppermost row in a column is row 1.

tbl_col_celllist

`count = tbl_col_celllist(columnId, array)`

This function fills *array* with the ID of each cell in column *columnId* from top to bottom and returns the number of IDs.

tbl_col_count

`tbl_col_count ([oid])`

This function returns the number of columns in the table given by object identifier *oid*. If *oid* is omitted, this function returns the number of columns in the table containing the cursor. Returns —1 if the cursor is not inside a table.

tbl_col_index

`index = tbl_col_index(columnId)`

Returns the position of column *columnId* within its grid. The leftmost column in a grid has index 1.

tbl_col_neighbor

`columnId = tbl_col_neighbor(columnId, offset)`

Returns the ID of the column *offset* columns to the left (*offset*<0) or right (*offset* >0) of column *columnId*.

tbl_col_rulelist

`count = tbl_col_rulelist(columnId, direction, array)`

This function fills *array* with the IDs of the rules on the edge *direction* of the column *columnId*.

tbl_coltool_mouse

`columnId = tbl_coltool_mouse(viewid)`

This function returns the ID of the column with which the mouse pointer is vertically aligned in view *viewId* (or the active view if *viewId* is omitted). It returns *h::tblNullToid* if the view doesn't contain an active table or if the mouse pointer isn't in a table column.

tbl_dlg_target

tbl_dlg_target ([*type*[, *doc*]])

This function returns the table object that table-related dialog boxes (such as the **Table Properties** dialog box) operate on. The return value is either a table selection object if one or more cells are selected, the ID of the cell containing the caret if there is no table selection, or `h::tblNullToid` if the caret is outside a table.

- *type* — Optional. Determines the type of table object returned and is one of the strings returned by the `tbl_obj_type` function. Values include `cell`, `row`, `column`, `grid`, `set`, and `selector`. For example, `tbl_dlg_target("row")` returns the row id of the cell containing the caret, or the row id of the first cell in the table selection.
- *doc* — Optional. Specifies the identifier of the document containing the table. If omitted or `0`, the current document is used.

tbl_edit_close

1 = **tbl_edit_close** (*success, [doc]*)

This function decrements the table-edit count for document *doc* (or the active document, if *doc* is omitted). If the count is zero and *success* is also zero, it calls `undo` to discard any changes to the document since the corresponding call to `tbl_edit_open` on page 549.

This function always returns 1.

Caution

Follow every call to `tbl_edit_open` with a call to `tbl_edit_close`. Failure to do so may cause Arbortext Editor to operate unexpectedly until restarted.

tbl_edit_open

tbl_edit_open (*toid*[, *doc*[, *undolabel*]])

This function increments the table-edit count for document *doc* (or for the active document if *doc* is omitted). This function returns `1` if it is completed successfully.

If *toid* is not equal to *h::tblNullToid*, it is used to indicate how much information Arbortext Editor must rescan if `tbl_edit_close` on page 549 is used to `undo` document changes.

The optional string argument *undolabel* is used as the name of the operation for the **Undo** and **Redo** menu items. The function returns a `0` if the *undolabel* refers to a cancelled operation that cannot be tracked within a *doc*.

Example

```
tbl_edit_open(toid, doc, "Insert Row");
```

Caution

Follow every successful call to `tbl_edit_open` with a call to `tbl_edit_close`. Failure to do so may cause Arbortext Editor to operate unexpectedly until restarted.

tbl_grid_cell

`cellId = tbl_grid_cell(gridId, colIndex, rowIndex)`

This function returns the ID of the cell whose position is given by *colIndex* and *rowIndex* in grid *gridId*. The upper-left cell in the grid has coordinates (1,1).

tbl_grid_celllist

`count = tbl_grid_celllist(gridId, array)`

This function fills *array* with the IDs of each cell in grid *gridId* and returns the number of IDs. The cell IDs are returned in row-major order.

tbl_grid_col

`columnId = tbl_grid_col(gridId, colIndex)`

This function returns the ID of column *colIndex* in grid *gridId*. The left-most column has an index of 1.

tbl_grid_colcount

`count = tbl_grid_colcount(gridId)`

This function returns the number of columns in grid *gridId*.

tbl_grid_collist

`count = tbl_grid_collist(gridId, array)`

This function fills *array* with the IDs of the columns in grid *gridId* ordered from left to right and returns the number of IDs.

tbl_grid_first_galley_cell

`cellId = tbl_grid_first_galley_cell(gridId)`

This function returns the first cell in grid *gridId* in galley order.

tbl_grid_insert

`gridId = tbl_grid_insert(gridId, addBefore, rowCount, columnCount)`

This function inserts a new grid with *rowCount* rows and *columnCount* columns into the table containing grid *gridId*. If *addBefore* is nonzero, the new grid is inserted before *gridId*. It returns the ID of the new grid or *h::tblNullToid* upon failure.

tbl_grid_last_galley_cell

`cellId = tbl_grid_last_galley_cell(gridId)`

This function returns the last cell in grid *gridId* in galley order.

tbl_grid_neighbor

`gridId = tbl_grid_neighbor(gridId, offset)`

This function returns the ID of the grid located *offset* grids before or after grid *gridId* in the table containing *gridId*.

tbl_grid_row

`rowId = tbl_grid_row (gridId, rowIndex)`

This function returns the ID of the row located *rowIndex* rows from the top of grid *gridId*. The uppermost row in any grid has an index of 1.

tbl_grid_rowcount

`count = tbl_grid_rowcount(gridId)`

This function returns the number of rows in grid *gridId*.

tbl_grid_rowlist

`count = tbl_grid_rowlist(gridId, array)`

This function fills *array* with the IDs of the rows in grid *gridId* ordered from top to bottom and returns the number of IDs.

tbl_grid_rule

`ruleId = tbl_grid_rule(gridId, startColIndex, startRowIndex, endColIndex, endRowIndex)`

This function returns the ID of the rule that connects the vertices (*startColIndex*, *startRowIndex*) and (*endColIndex*, *endRowIndex*) in grid *gridId*.

tbl_grid_rulelist

`count = tbl_grid_rulelist(gridId, array)`

This function fills *array* with the IDs of the rules in grid *gridId* sorted into row-major order and returns the number of IDs.

tbl_grid_split

`[0|1] = tbl_grid_split(toid)`

This function splits the grid containing *toid* into two separate grids, with the row containing *toid* as the uppermost row in the second grid. It fails if the table model doesn't support multiple grids in the same table or if performing the split would result in either or both grids being invalid in any way according to the table model's rules. *toid* may be the ID of either a row or a cell.

tbl_hline_rulelist

`count = tbl_hline_rulelist(gridId, y, x1, x2, array)`

This function fills *array* with the IDs of the rules occupying the horizontal line identified by the coordinates *y*, *x1*, and *x2* in grid *gridId* ordered from left to right and returns the number of IDs.

tbl_insert

`id = tbl_insert (tmid, oid, pos, rows, cols, [tagname])`

This function inserts a new table, specified by table model *tmid*, with *rows* rows and *cols* columns, into the document at the point indicated by *oid* and *pos*. If *tmid* is zero (0), then the primary table model for the document type will be used.

tagname is only used by document types that use wrappers. Some document types allow several different tags to serve as the wrapper tag surrounding a table. *tagname* is used to specify a wrapper tag; if *tagname* is omitted, the outer wrapper tag is chosen at random.

For custom tables, the *cols* value is ignored.

Note

If you have applied an alias map to the document, *tagname* can be an alias, as well as a real name.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

It returns the ID of the newly-inserted table set if successful, *h::tblNullToid* otherwise.

The `tbl_insert` function will not do a pending delete regardless of the setting of `set pendingdelete` on page 866.

tbl_insert_rows_cols_dlg

`[0|1] = tbl_insert_rows_cols_dlg([win])`

This function displays the **Insert Rows and Columns** dialog box, which attempts to insert a user-specified number of rows and columns at the cursor in the document shown in window *win* (or in the active window if *win* is omitted). It returns 0 if no table is active. It returns 1 on success.

tbl_insert_table_dlg

`[0|1] = tbl_insert_table_dlg([win])`

This function displays the **Select a Table Model** dialog box in which you can choose a table model if the following conditions are true:

- If your document type supports multiple table models.
- `set prompttablemodels` is set to `on`.
- The cursor is in a location in which it is valid to insert more than one table model.

Once you have chosen a table model, or, if your DTD doesn't support multiple table models, the **Insert Table** dialog box opens, in which you can specify table configuration options.

The function then attempts to insert the table at the cursor location in the document shown in window *win* (or in the active window if *win* is omitted).

The function returns `0` if a table cannot be inserted at the cursor location. It returns `1` if a table is inserted.

tbl_insertion_valid

`[0|1} =` **`tbl_insertion_valid`**`([`*`doc`*`])`

This function returns 1 if a table can be inserted at the cursor point in document *doc* (or in the active document if *doc* is omitted).

If context checking is turned off, this function returns 1 regardless of where the cursor is located in the document. If `set autotaginserts` on page 754 is set to `on`, this function returns 1 if the cursor is at a point in the document where a tag can be automatically inserted that would then make the insertion of a table valid.

tbl_mod_borders_dlg

`[0|1] =` **`tbl_mod_borders_dlg`**`(`*`[win]`*`)`

This function displays the **Table Properties** dialog box and attempts to modify the borders of the active table in window *win* (or in the active window if *win* is omitted). It returns 0 if the window contains no active table. It returns 1 on success.

tbl_mod_cellfont_dlg

`[0|1] =` **`tbl_mod_cellfont_dlg`**`(`*`[win]`*`)`

This function displays the **Font** dialog box and attempts to modify the cell font of the active table in window *win* (or in the active window if *win* is omitted). It returns 0 if the window contains no active table. It returns 1 on success.

tbl_mod_cells_dlg

`[0|1] = tbl_mod_cells_dlg([win])`

This function displays the **Table Properties** dialog box and attempts to modify the cell attributes of the active table in window *win* (or in the active window if *win* is omitted). It returns 0 if the window contains no active table. It returns 1 on success.

tbl_mod_table_dlg

`[0|1] = tbl_mod_table_dlg([win])`

This function displays the **Table Properties** dialog box and attempts to modify the table attributes of the active table in window *win* (or in the active window if *win* is omitted). It returns 0 if the window contains no active table. It returns 1 on success.

tbl_model_celllist

`count = tbl_model_celllist(tmid, array, [hdr])`

This function fills *array* with the names of the cell tags used in the table model whose ID is *tmid*. If *hdr* is supplied and is non-zero, then the names of the header cell tags will be returned in *array*.

If *tmid* is invalid, the function returns a zero (0); otherwise, it returns the number of cells tags in *array*.

tbl_model_id

`tmid = tbl_model_id (name[, doc])`

This function returns the table model ID (tmid) corresponding to the table model *name* (`OASIS Exchange`, `ATI`, or `HTML`) for the document specified by *doc*. If *doc* is omitted or 0, the current document is used.

If *name* is a null string, the primary table model configured for the document is used.

The `tbl_model_id` function returns `-1` if no tables are configured for the document.

tbl_model_list

`count = tbl_model_list(array, [doc])`

This function fills *array* with the IDs of the table models valid for document *doc* (or for the active document if *doc* is omitted) and returns the number of IDs. It returns 0 if *doc* is invalid or if *doc* has no table models.

tbl_model_name

`return = tbl_model_name(tmid)`

This function returns the name of the table whose table model ID is indicated by *tmid*. The table model name may be one of: “OASIS Exchange,” “ATI,” “HTML”, “XSL-FO”, or if the table is a custom table, the name specified in the document type configuration file (.dcf) from the attribute `name` on the element `CustomTableModel`.

If *tmid* is invalid, the function returns the string ”unknown”.

tbl_model_operation

`result = tbl_model_operation(opcode, toid, ...)`

This function requests a table-model-specific function. The parameters and result value are determined by the operation requested by *opcode*.

- All operations are supported for tables managed by the OASIS Exchange table model.
- No operations are supported for tables managed by the HTML, XSL-FO, or PTC Arbortext table models.

toid specifies different items based on the value of *opcode*. *opcode* can be `0`, `1`, `2`, `3`, `4`, or `5`.

- If *opcode* is `0`, all `colspec` tags within `thead` or `tfoot` tags are deleted, and every `entry` tag in the table is adjusted to refer to the `colspec` tags that are children of the `tgroup` tag.

  When *opcode* is `0`, *toid* can specify any of the following items and resulting actions by Arbortext Editor:

  - A selection object — Arbortext Editor processes every object.
  - A table set (TSet) — Arbortext Editor processes every grid in the set.
  - A grid (TGrid) — Arbortext Editor processes the corresponding TGroup.
  - A row, column, or cell — Arbortext Editor processes the containing TGrid.
- If *opcode* is `1`, Arbortext Editor deletes `spanspec` tags and updates `entry` tags to refer to `colspec` tags. If the document type does not allow *namest* or *nameend* attributes on `entry` tags, `tbl_model_operation` does nothing.

When *opcode* is `1`, the *toid* parameter is interpreted in the same manner as when *opcode* is `0`.

- If *opcode* is `2`, Arbortext Editor updates the *colname* attribute of every `colspec` tag in a table.

  When *opcode* is `2`, the *toid* parameter is interpreted in the same manner as when *opcode* is `0`.

  When *opcode* is `2`, `tbl_model_operation` takes up to 3 additional parameters: *colroot*, *leftcolspec*, and *increment*.

  - *colroot* is a quoted string containing the characters `%d` specifying the root column name. (For example, `"column%d"`.) `%d` is replaced with `0` for the left-most column, `1` for the next left-most, and so on. If *colroot* does not contain `%d`, `%d` is appended to the specified string. If *colroot* is omitted, `"col%d"` is used.
  - *leftcolspec* specifies the number to be used for the left-most `<colspec>` in the table. If *leftcolspec* is omitted, `0` is used.
  - *increment* specifies the increment to be used during the renaming process. If *increment* is omitted, `1` is used.

  For example, a subroutine call such as

  ```
  $x=tbl_model_operation(2, toid, "column%d", 5, 2)
  ```

  causes Arbortext Editor to rename the `<colspec>` tags to `column5`, `column7`, `column9`, and so on.

- If *opcode* is `3`, Arbortext Editor scans a table and reorganizes the attributes of the various table tags to minimize the number of attributes required to describe the table.

  When *opcode* is `3`, *toid* specifies a TSet or any table object. Arbortext Editor processes the TSet.

- If *opcode* is `4`, Arbortext Editor renames a single `colspec` tag by updating the tag's *colname* attribute and adjusting all `spanspec` and `entry` tags to refer to the `colspec` by its the new value for the *colname* attribute.

  When *opcode* is `4`, the *toid* parameter is interpreted in the same manner as when *opcode* is `2`.

  When *opcode* is `4`, `tbl_model_operation` takes up to 3 additional parameters: *colexam*, *oldspecname*, and *newspecname*.

  - *colexam* is a numeric value of 0, 1, 2, 3, or 4.
    - *0* — Arbortext Editor examines all `colspec` tags.
    - *1* — Arbortext Editor examines only the `colspec` tags in a `thead`.

- *2* — Arbortext Editor examines only the `colspec` tags in a `tfoot`.
- *3* — Arbortext Editor examines only the `colspec` tags in the top level in a `tgroup`.

- *oldspecname* is a quoted string specifying the `colspec` to be renamed. (That is, the `colspec` with `colname=`*`oldspecname`*.)
- *newspecname* is a quoted string specifying the new name of the `colspec`.

For example, a subroutine call such as

```
$x=tbl_model_operation(4, toid, 0, "col1", "column1")
```

causes Arbortext Editor to rename any `colspec` named `col1` to be named `column1`, and to update all `spanspec` and `entry` tags that referred to the old `colspec` name.

If no `colspec` has the name specified by *oldspecname*, `tbl_model_operation` does nothing. If another `colspec` already has the name specified by *newspecname*, `tbl_model_operation` generates an error.

- If *opcode* is `5`, Arbortext Editor renames a single `spanspec` tag by adjusting the tag's *spanname* attribute and modifying every `entry` tag that refers to it.

  When *opcode* is `5`, the *toid* parameter is interpreted in the same manner as when *opcode* is `2`.

  When *opcode* is `5`, `tbl_model_operation` takes 2 additional parameters: *oldspanname* and *newspanname*.

  - *oldspanname* is a quoted string specifying the `spanspec` to be renamed.
  - *newspanname* is a quoted string specifying the new name of the `spanspec`.

  For example, a subroutine call such as

  ```
  $x=tbl_model_operation(5, toid, "span1", "span2")
  ```

  causes Arbortext Editor to rename any `spanspec` named `span1` to be named `span2`, and to update all `entry` tags that referred to the old `spanspec` name.

  If no `spanspec` has the name specified by *oldspanname*, `tbl_model_operation` does nothing. If another `spanspec` already has the name specified by *newspanname*, `tbl_model_operation` generates an error.

tbl_model_row

`tagname = `**`tbl_model_row`**`(`*`tmid, [hdr]`*`)`

This function returns the tag name of the row tag used in the table whose table model ID is *tmid*. If *hdr* is supplied and is non-zero, then the name of the header row tag, if it exists, is returned instead.

If *tmid* is invalid or *hdr* is specified and there is no header row, then, the function returns an empty string.

tbl_model_support

`ret = tbl_model_support(tmid, opcode)`

This function returns 1 if the given table model id (*tmid*) supports the feature specified by the *opcode*. If it does not or if either parameter is invalid, the function returns a zero (0).

Valid *opcode* values are:

- 0 – multiple table grids in a single table set
- 1 – header rows
- 2 – footer rows
- 3 – background color on cells
- 4 – an attribute to set frame property
- 5 – attributes to set cell spacing and cell padding

tbl_model_table_title

`tagname = tbl_model_table_title(tmid)`

This function returns the tag name of the title or caption tag used in the table model whose ID is *tmid*.

If *tmid* is invalid, the function returns an empty string.

tbl_model_tablelist

`count = tbl_model_tablelist(tmid, array)`

This function fills *array* with the names of the table root tags used in the table model whose ID is *tmid*. The root tag is the grid tag for the table.

If *tmid* is invalid, the function returns a zero (0); otherwise, it returns the number of tags in *array*.

tbl_model_taglist

`count = tbl_model_taglist(tmid, array)`

This function fills *array* with the names of all the table tags used in the table model *tmid*, and returns the number of tags inserted in the array. If *tmid* is invalid, the function returns a zero (0).

This function does not return table wrapper tags; these are retrieved by the `tbl_model_wrapperlist` on page 560 function.

tbl_model_wrapperlist

`count = tbl_model_wrapperlist(tmid, array)`

This function fills *array* with the names of the wrapper tags used in the table model *tmid*, and returns the number of tags inserted in the array. If *tmid* is invalid, the function returns a zero (0).

tbl_multicell_border

`count = tbl_multicell_border(cellId, direction, outArray)`

This function fills *outArray* with the ID of each rule on side *direction* of multicell *cellId* sorted from top to bottom or left to right and returns the number of IDs.

tbl_multicell_celllist

`count = tbl_multicell_celllist(cellId, array)`

This function fills *array* with the ID of each cell in multicell *cellId* sorted into row-major order and returns the number of IDs.

tbl_multicell_neighborlist

`count = tbl_multicell_neighborlist(cellId, whichSide, outArray)`

This function fills *outArray* with the ID of each cell adjacent to edge *whichSide* of multicell *cellId* and returns the number of values placed in *array*. The cell IDs are returned in row-major order.

tbl_multicell_size

`[0|2] = tbl_multicell_size(cellId, array)`

This function returns the number of rows and columns occupied by multicell *cellId* in the output *array*. The first *array* element contains the row count, and the second *array* element contains the cell count. The function returns a value of 0 if the cell is either not in a multicell or the *cellId* is invalid. Otherwise, the function returns a value of 2.

The following example returns the number of rows or cells spanned by a given cell based on its `oid`. Note that this example might be used by a FOSI `system-func` call, so the `window` argument is required but not used.

```
function row_count(window,oid) {
 return span_count($window,$oid,1);
}
function col_count(window,oid) {
 return span_count($window,$oid,0);
}
function span_count(window,oid,rowchk) {
 local rowcount;
 local colcount;
 local size[];
 if (!tbl_cell_in_multicell(tbl_oid_cell($oid))) {
 return 1;
}
 tbl_multicell_size(tbl_oid_cell($oid), size);
 rowcount = size[1];
 colcount = size[2];
 if ($rowchk == 1) {
 return rowcount;
} else {
 return colcount;
}
}
```

tbl_multicell_spanner

`cellId = **tbl_multicell_spanner**(*cellId*)`

Returns the cell ID of the spanning cell of multicell *cellId*. *cellId* may indicate any cell in the multicell.

tbl_obj_add

`toId = **tbl_obj_add** (*toId*, *addBefore*, *[tagname]*)`

This function inserts a new row or column, of the same type as table object *toid*, before or after *toid* as indicated by a nonzero value in *addBefore*. It returns the ID of the newly-inserted row or column.

For custom tables, when multiple elements are acceptable to be used as the row element, if the *tagname* parameter is supplied, then the *tagname* element is inserted.

tbl_obj_attr_clear

`[0|1] = tbl_obj_attr_clear(toid)`

This function deletes all of table object *toid*'s attribute values by resetting the table markup attributes from which they're derived to default values.

If *toid* is invalid, the function returns a zero (`0`). If the function executes successfully, it returns a one (`1`).

The *toid* parameter is the table object ID of the table object whose attributes are to be cleared.

tbl_obj_attr_delete

`[0|1] = tbl_obj_attr_delete(toid, attrName)`

This function deletes table object attribute *attrName* from table object *toid* by resetting the markup from which the attribute was derived to a default value.

If *toid* or *attrName* is invalid, the function returns a zero (`0`). If the function executes successfully, it returns a one (`1`).

The *toid* parameter is the table object ID of the table object whose attribute value is to be deleted. The *attrName* parameter is a string for the name of the attribute. Note that names are case insensitive.

tbl_obj_attr_get

`[0|1] = tbl_obj_attr_get(toid, attrName, attrval)`

This function places the value of table object attribute *attrName* for table object *toid* in *attrval*.

If *toid* or *attrName* is invalid, the function returns a zero (`0`). If the function executes successfully, it returns a one (`1`).

The *toid* parameter is the table object ID of the table object whose attribute value is to be retrieved. The *attrName* parameter is a string for the name of the attribute. Note that names are case insensitive. The *attrval* parameter is a variable in which the attribute value is returned. Some values are numbers, some are strings. String values are case insensitive.

tbl_obj_attr_set

`[0|1] = tbl_obj_attr_set(toid, attrName, attrval)`

This function sets table object attribute *attrName* for table object *toid* to *attrval*.

If *toid* or *attrName* is invalid, the function returns a zero (`0`). If the function executes successfully, it returns a one (`1`).

The *toid* parameter is the table object ID of the table object whose attribute value is to be set. The parameter is a string for the name of the attribute. Note that names are case insensitive. The *attrval* parameter is the new value for the attribute *attrName*. Some values are numbers, some are strings. String values are case insensitive.

tbl_obj_attr_valid

`[0|1] = tbl_obj_attr_valid(toid, attrName[, attrval])`

This function determines if setting table object attribute *attrName* for table object *toid* is valid or if setting table object attribute *attrName* for table object *toid* to *attrval* is valid.

If *toid* or *attrName* is invalid, or setting the table object attribute is not valid, `tbl_obj_attr_valid` returns zero (`0`). If setting the table object attribute is valid, `tbl_obj_attr_valid` returns one (`1`).

- *toid* — The table object ID of the table object whose attribute value is to be set.
- *attrName* — A string for the name of the attribute. Names are case insensitive.
- *attrval* — Optional. The new value for the attribute *attrName*. Values are numbers or strings. String values are case insensitive. If *attrval* is not supplied, the function tests whether the table model supports setting the *attrName* table object attribute at all. If *attrval* is supplied, then the function is testing whether the table model supports setting the *attrName* table object attribute to the specific value.

tbl_obj_delete

`[0|1] = tbl_obj_delete(toId)`

This function deletes table object *toid* from its document. *toid* may indicate a table set, a grid, a row, or a column.

tbl_obj_grid

`gridId = tbl_obj_grid(toId)`

This function returns the ID of the grid containing table object *toId*.

tbl_obj_insert

`[0|1] = tbl_obj_insert(ulSourceCellId, lrSourceCellId, targetToid, addBefore)`

This function inserts a new grid, row rectangle, or column rectangle into an existing set or grid and copies the contents of the source rectangle into the new one. The source rectangle is defined by upper-left corner cell *ulSourceCellId* and lower-right corner cell *lrSourceCellId*; the target grid, row, or column is identified by *targetToid*. The width or height of the source rectangle must specify one or more complete rows or columns if *targetToid* is a row or a column.

tbl_obj_mark

`[0|1] = tbl_obj_mark (toId, newRegion, primarySelection, doInvert)`

This function begins or ends a mark operation on a rectangle of table objects.

toId: the table object id of the table object to start or end the mark on.

newRegion: [0|1]. If non-zero, start a separate table object selection. Only relevant on 3rd and successive odd numbered calls to `tbl_obj_mark`.

primary: may be 1, 0, or -1, to set, clear, or leave alone the primary selection.

doInvert: [0|1]. If non-zero, invert the selection in all views.

This function implements a finite state machine with three states: empty, marking, and bound. We start empty; the first call to `tbl_obj_mark` saves *toid* as the selection *anchor* and changes state to *marking*. The second call to `tbl_obj_mark` saves *toid* as the selection endpoint and changes state to *bound*.

A third call to `tbl_obj_mark` discards the previous endpoint and switches back to *marking* state; a fourth call switches back to *bound*.

Issuing the *clear_mark* command is the only way to get back to *empty* state.

If our state is *marking* or *bound*, we know two points: the *anchor* and the *endpoint*. Unless these two points are identical, we can invert the display region between them to clue the user in about what's marked.

`tbl_obj_mark` allows discontiguous rectangles of table objects to be selected. We handle discontiguity by checking for the *newRegion* flag on odd-numbered calls (state == empty or state == bound) and pushing the anchor and end points onto a stack before starting a new mark operation.

While our state is *marking*, the user may adjust the region endpoint by calling `tbl_obj_markdrag`; the state stays at *marking*, but the point (and hence the inverted region) changes.

Note that calling `tbj_obj_mark` will clear any text selection, and that selecting text will clear any table object selection.

Note that invoking the *mark* command also forces a state transition.

Returns 1 if successful. 0 (indicating failure) is returned when *TOId* doesn't identify a valid table object.

tbl_obj_markdrag

`[0|1] =` **`tbl_obj_markdrag`**`(`*`toId`*`)`

This function changes the endpoint of the currently-active selection rectangle to table object *toid*. It returns 1 if a table selection was defined and its endpoint successfully changed.

tbl_obj_marked

`[0|1] =` **`tbl_obj_marked`**`(`*`toId`*`)`

This function returns 1 if table object *toid* is selected (that is included in the selection object belonging to its document).

tbl_obj_modifiable

`[0|1] =` **`tbl_obj_modifiable`**`(`*`toId`*`)`

This function returns 1 if table object *toid* is unprotected.

tbl_obj_set

`setId =` **`tbl_obj_set`** `(`*`toId`*`)`

This function returns the id of the table set that contains table object *toId*.

tbl_obj_type

`objectType =` **`tbl_obj_type`**`(`*`toId`*`)`

This function returns the object type of table object *toId*.

tbl_obj_valid

`[0|1] = tbl_obj_valid(toId)`

This function returns 1 if table object *toId* is valid.

tbl_obj_viewimage

`[0|1] = tbl_obj_viewimage(id, [win])`

This function returns 1 if table object *id* is part of a table image in window *win* (or in the active window, if *win* is omitted).

tbl_oid_cell

`id = tbl_oid_cell (oid, [pos])`

This function returns the ID of the cell containing the point indicated by *oid* and *pos*. If the point is not in a table cell, it returns *h::tblNullToid*.

tbl_oid_nodelete

`[0|1] = tbl_oid_nodelete(oid)`

This function returns 1 if the table model managing the tag indicated by *oid* should be protected from deletion. This allows GUI elements to delete the content of a cell without deleting PIs and other things that a table model might care about.

tbl_oid_object

`toid = tbl_oid_object(oid, [pos])`

This function returns the toid of the deepest table object (a cell, row, grid, or set) that fully contains the specified *oid* and optional parameter *pos*. If the specified combination of *oid* and *pos* is not inside table markup, the function returns `h::tblNullToid`.

tbl_oid_viewimage

`[0|1] = tbl_oid_viewimage(oid, [win])`

This function returns 1 if *oid* is contained in a table that is displayed as an image (instead of as tags) in window *win* (or in the active window if *win* is omitted.

tbl_row_cell

`cellId = tbl_row_cell(rowId, index)`

This function returns the ID of cell *index* in row *rowId*. The leftmost cell in a row has an index of 1.

tbl_row_celllist

`count = tbl_row_celllist(rowId, array)`

This function fills *array* with the IDs of the cells in row *rowId* ordered from left to right and returns the number of IDs.

tbl_row_count

`count = tbl_row_count([oid])`

This function returns the number of rows in the table image given by object identifier *oid*. If *oid* is omitted, it returns the number of rows in the table containing the cursor. Returns —1 if the cursor is not inside a table.

tbl_row_index

`index = tbl_row_index(rowId)`

This function returns the index of row *rowId* relative to the top row in the containing grid. The uppermost row in a grid has an index of 1.

tbl_row_neighbor

`rowId = tbl_row_neighbor(rowId, offset)`

This function returns the ID of the row *offset* rows above (*offset*<0) or below (*offset*>0) row *rowId*.

tbl_row_rulelist

`count = tbl_row_rulelist(rowId, direction, array)`

This function fills *array* with the IDs of the rules in row *rowId* in direction *direction*. The IDs are ordered from left to right. It returns the number of IDs.

tbl_rowtool_mouse

`rowId = tbl_rowtool_mouse(viewid)`

This function returns the ID of the row in the active grid in view *vidId* (or the active view, if *viewId* is omitted) that contains the mouse pointer.

tbl_rule_cellneighbor

`cellId = tbl_rule_cellneighbor(ruleId, direction)`

This function returns the ID of the cell bordering rule *ruleId* on side *direction*.

tbl_rule_is_suppressed

`[0|1] = tbl_rule_is_suppressed(ruleId)`

This function returns 1 if rule *ruleId* is suppressed because it is inside a multicell.

tbl_rule_orientation

`orientation = tbl_rule_orientation(ruleId)`

This function returns the orientation of rule *ruleId*.

tbl_rule_ruleneighbor

`ruleId = tbl_rule_ruleneighbor(ruleId, direction, offset)`

This function returns the ID of the rule *offset* cells away from rule *ruleId* in direction *direction*.

tbl_rule_vertices

`[0|1] = tbl_rule_vertices(ruleId, startColIndex, startRowIndex, endColIndex, endRowIndex)`

This function places the coordinates of the start and end points of rule *ruleId* in the output variables *startColIndex*, *startRowIndex*, *endColIndex* and *endRowIndex*.

tbl_selection_clone

`id = tbl_selection_clone(selectionId)`

This function returns the ID of a new table selection object that contains the same table object IDs as table selection object *selectionId*. The user is responsible for freeing the object by calling `tbl_obj_delete`.

This routine can be used in conjunction with `tbl_selection_restore` and in circumstances where executing some ACL command would result in prematurely clearing a document's table object selector.

tbl_selection_empty

`[0|1] = tbl_selection_empty(selectionId)`

This function returns 1 if the table selection object *selectionId* contains no table object IDs.

tbl_selection_get

`id = tbl_selection_get([doc])`

This function returns the ID of the selection object used to manage table selections in the indicated document (or in the active document if *doc* is omitted).

tbl_selection_matchbegin

`[0|1] = tbl_selection_matchbegin(selectionId, wantSets, wantGrids, wantColumns, wantRows, wantCells, wantRules, contiguous, preferColumns)`

This function organizes the contents of selection object *selectionId* and prepares for subsequent calls to `tbl_selection_matchnext`. A selection object contains a series of rectangles, each of which might cover an entire grid, one or more entire rows or columns, or simply a smaller area within a single grid. At different times, it may be advantageous to iteratively examine every cell in every rectangle; at others, a user might want to require ensure that only complete rows are selected, and iteratively process each row. `tbl_selection_matchbegin` supports all such possibilities by analyzing the selection object and organizing the rectangles it contains into the largest requested units. It returns '0' if the selection object contains a rectangle that can't be so organized, and '1' if the selection object contains rectangles that can be interpreted as the user indicates. In the latter case, calls to `tbl_selection_matchnext` can be used to retrieve the interpreted table objects.

tbl_selection_matchnext

`toid = tbl_selection_matchnext(selectionId)`

This function supports iteration over the contents of selection object *selectionId* by returning the Table Object ID (toid) of the next table object matching the search criteria specified on the most recent call to `tbl_selection_matchbegin` on page 569. If this function is used immediately after a call to

`tbl_selection_matchbegin`, it will find the first match (that is, the match found by `tbl_selection_matchbegin`). A subsequent call to this function will find the next match.

If there is no such object, it returns the value of *h::tblNullToid*.

This function searches for Table Object IDs per the following rules:

- Row, column, and cell toids from a single rectangle are returned in row-major order.
- Rule toids are returned in row major order.
- Set toids are returned in no particular order.

tbl_selection_nextrectangle

`[0|1] =` **`tbl_selection_nextrectangle`**`(`*`selectionId`*`, `*`startFlag`*`, `*`ulCellId`*`, `*`lrCellId`*`)`

This function iterates over the contents of selection object *selectionId*, returning in the output variables *ulCellId* and *lrCellId* the Cell IDs of the upper-left and lower-right corners of the first or next rectangle (*startFlag* = 1 or 0, respectively) within. It returns '0' if there is no first or next rectangle, '1' otherwise.

tbl_selection_pasterectangle

`[0|1] =` **`tbl_selection_pasterectangle`**`(`*`selectionId`*`)`

This function returns '1' if the selection object indicated by *selectionId* contains TOIds that describe a single rectangle.

The rectangle of cells identified this way is suitable for use as the target of a paste operation.

tbl_selection_pastetype

`type =` **`tbl_selection_pastetype`**`(`*`selectionId`*`, `*`targetId`*`)`

This function indicates whether pasting the table objects whose IDs are contained in selection object *selectionId* into the destination cell, row, column, or grid indicated by *targetId* will replace an entire grid, one or more entire rows, one or more entire columns, or just a rectangle of cells.

This routine returns h::tblTypeUndef (undefined type) if either parameter is invalid, if the selection object doesn't contain TOIds describing a single rectangle, or if the targetId indicates a table object not contained in a grid.

tbl_selection_restore

`[0|1] = tbl_selection_restore(selectionId)`

This function selects the table objects whose IDs are contained by the selection object*selectionId*.

tbl_selection_tmid

`tmid = tbl_selection_tmid(selectionId)`

This function returns the ID of the table model responsible for managing the table objects whose IDs are contained in selection object *selectionId*.

tbl_selection_valid

`[0|1] = tbl_selection_valid(selectionId)`

This function returns 1 if all of the table object IDs in selection object *selectionId* are valid.

tbl_set_first_galley_cell

`toid = tbl_set_first_galley_cell(SetId)`

This function returns the ID of the first cell in set *setid* in galley order.

tbl_set_grid

`gridId = tbl_set_grid(setId, gridIndex)`

This function returns the ID of grid number *gridIndex* in set *setId*. The first grid in a set is grid number 1.

tbl_set_gridlist

`count = tbl_set_gridlist(setId, array)`

This function fills *array* with the IDs of each grid in set *setId* and returns the number of IDs.

tbl_set_last_galley_cell

`toid = tbl_set_last_galley_cell(setId)`

This function returns the ID of the last cell in set *setId* in galley order.

tbl_table_title_delete

`[0|1] = tbl_table_title_delete(toidId)`

This function deletes the table title (caption) from the table containing *toidId*. The function returns `1` if a table title existed and was deleted. Otherwise, `tbl_table_title_delete` returns `0`.

tbl_table_title_insert

`oid = tbl_table_title_insert(toidId[, str])`

This function inserts the table title (caption) *str* into the table containing *toidId*. The function returns the oid of the new table title if one was inserted, or it returns the oid of a previously existing table title. If the table's table model does not allow table titles or the insert otherwise fails, the function returns `null_oid`.

The HTML table model supports table titles with the `caption` element. Custom table models may be configured to support table titles. The CALS and Arbortext table models do not support table titles.

tbl_vline_rulelist

`count = tbl_vline_rulelist(gridId, x, y1, y2, array)`

This function fills *array* with the IDs of the rules that occupy the vertical line identified by the coordinates *x*, *y1*, and *y2* in grid *gridId* and returns the number of IDs.

tell

`offset = tell (fid)`

This function returns the offset of the current byte relative to the beginning of the file associated with the identifier *fid*, which must be a return value from a previous call to `open`.

`tell` returns `-1` on failure, for example, if *fid* is not a valid file identifier.

temp_name

`$tempfile = temp_name(name[, ext[, dir]])`

This function creates a new temporary log file. The *name* parameter specifies the base name for the file. The optional *ext* parameter specifies the extension to use for the file name. If the *ext* parameter is omitted, it defaults to `.log`.

The optional *dir* parameter specifies the where to store the new temporary file. If the *dir* parameter is omitted, it defaults to the value of the environment variable *TMPDIR*. If *TMPDIR*, is not set, it defaults to the `c:\temp` directory.

The return value is the name of the newly created temp file. Note that this function will automatically append a number (1 to 99) to the end of the base file name as necessary to avoid duplicate file names.

Be aware that another process could feasibly create a file with the same name in the time between when this function is called and when the file is actually created.

terminal_mode

`[0|1] =` **`terminal_mode`**`([`*`flags`*`])`

This function determines whether windows are available in Arbortext Editor. If the *type* is not supplied or is `0`, this function returns `1` if windows are not available or `0` if windows are available. Windows are not available if Arbortext Editor is run in `-c` (command) mode or `-S` (server) mode. To test for a specific non windows mode, then type should be set to `2` for `-c` (command) mode or `3` for `-S` (server) mode.

text_entity_names

`count =` **`text_entity_names`**`(`*`arr`*`[, `*`doc`*`])`

The `text_entity_names` function fills the array *arr* with an alphabetical list of text entity names which are valid for the current document, returning the total number. The first name returned is stored at index 1. All previous elements in *arr* are discarded.

The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

Note

Entities declared in the DTD are included in the array *arr* in addition to those declared in the declaration subset.

text_entity_tag

`[0|1] =` **`text_entity_tag`**`(`*`tagname`*`[, `*`doc`*`])`

This function returns `1` (True) if the tag specified by the expression *tagname* is really a tag used to represent an SGML text entity declaration. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

text_style_tag_name

`tagname = **text_style_tag_name**(*style*, *arr*[, *doc*])`

This function returns the name of the primary element for the given *style* specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. Valid *style* values are `bold`, `italic`, and `underline`.

If a tag name is returned, `text_style_tag_name` function returns *arr* with an attribute name and attribute value, if they were specified for the tag in the `.dcf` file.

The function returns the null string if there is no element defined for the specified *style*, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

For example,

```
local tag, arr[];
local attr, val;
tag = text_style_tag_name("bold", arr);
attr = arr['attribute'];
val = arr['attribute_value'];
```

text_style_tag_name_ns

`tagname = **text_style_tag_name_ns**(*style*, *arr*[, *doc*])`

This function returns the namespace URI prefixed to the primary element for the given *style* specified in the document type configuration file (`.dcf`) for the document type associated with *doc*. Valid *style* values are `bold`, `italic`, and `underline`.

If a tag name is returned, `text_style_tag_name_ns` function returns *arr*.

The function returns the null string if there isn't a namespace prefix, if no element is defined for the specified *style*, or if *doc* is invalid or an ASCII file. If *doc* is omitted or `0`, the current document is used.

thesaurus

thesaurus ([*doReplace*[, *word*]])

This function opens the Thesaurus dialog box, using the selected word in the active document as the default value of *word*. If no word is selected, the word nearest to the cursor is used.

The function looks up *word* in the **Looked Up:** box. If *doReplace* is set to a non-zero value (or is not specified), clicking **Replace** on the dialog box will substitute the preferred word. If *doReplace* is set to zero (`0`), clicking **Replace** will close the dialog box without performing a word replacement.

throw

`throw``(`*`expr`*`)`

This function transfers control by returning from the innermost active `catch` function. The value of the expression *expr* becomes the return value from `catch`. *expr* must not evaluate to zero; if it does, 1 is used instead.

If `throw` is used outside the context of a `catch`, control is transferred to the top level, aborting the command execution.

time (Function)

`seconds = `**`time`**` ()`

This function returns the current time in seconds since the epoch, which is operating system-specific. This time can be converted into a string for display using the `ctime` function.

Example

```
ctime(time())
```

is equivalent to

```
time_date()
```

time_date

`string = `**`time_date`**` ([`*`gmt`*`])`

This function returns the current time as a string in the following format:

```
Mon Dec 28 17:23:52 1992
```

All the fields have constant width.

The argument `gmt` is optional. If non-zero, the time returned is in GMT. The local time is returned if the argument is omitted or zero. Note that the string returned does not include time zone information.

times

`time = **times** ([arr])`

The `times` function fills the array *arr* with four elements giving the user and system CPU times, in 1/100th seconds, for this process and the children of this process. It returns the sum of *arr* [1] and *arr* [2], that is, the total user and system time for the process. If *arr* is not given, then only the total time is returned. For example, to time a section of Arbortext Command Language code:

```
times($t1)
# whatever code here
times($t2)
$tdiff = $t2[1] - $t1[1]
eval "user time:", $tdiff/100 . '.' . $tdiff%100
$tdiff = $t2[2] - $t1[2]
eval "system time:", $tdiff/100 . '.' . $tdiff%100
```

tolower

`string = **tolower** (string)`

This function returns the string specified by *string* with all letters converted to lowercase.

toupper

`string = **toupper** (string)`

This function returns the string specified by *string* with all letters converted to uppercase.

treeloc_oid

`oid = **treeloc_oid** (treeloc[, sep_char[, top_level_oid]])`

This function returns the *oid* in the current document of the tag specified by *treeloc*.

Specify a separator character *sep_char* to delineate the values in the string with a character other than “,”, which is the default.

To limit the size of the returned value, specify a *top_level_oid*. This sets the parent tag at which the *oid* will begin. If you do not specify a *top_level_oid*, the first value in the returned string represents the first tag in the document.

Note

If you specify an *treeloc* within a file entity, this function will not search back into the parent document .

trim

`string = trim (string)`

This function returns the string specified by *string* with leading and trailing white space removed.

truncate

`[0|-1] = truncate(fid, length)`

This function sets the size of the file identified by *fid* to *length* bytes. *fid* must refer to a file that was opened for writing. If the file is currently larger than *length*, the extra bytes are removed. If the file is shorter, then zero bytes are added. The function returns `0` on success or `-1` on failure.

ucstombs

`return = ucstombs (str[, charset])`

This function converts the Unicode string *str* to a byte string in the specified character charset, which may be a multi-byte character set. *charset* is one of the character sets listed in the description of `mblen`. If *charset* is not specified, the system character set is assumed.

Because the resulting string is a byte string, not all string operations are appropriate. The result would normally be written to a binary file or network channel. See the example with `mbstoucs`.

Example

```
write(ch, ucstombs("get toc.htm\r\n", "iso8859-1"))
```

umask

`mask = umask ([newmask])`

This function sets the process's file creation mask to *newmask* and returns the previous value of the mask. *newmask* will affect all files created subsequently by Arbortext Editor, and will be inherited by any processes it spawns. If *newmask* is omitted, `umask` just returns the current mask.

undeclare_ms_parameter

`undeclare_ms_parameter``(`*`entity`*`)`

This function allows you to remove a marked section parameter entity declaration.

entity is the name of an existing marked section parameter entity that isn't referenced in the document.

undo_menu_description

`[0|1] =` **`undo_menu_description`**`(`*`label`*`[, `*`doc`*`])`

This function assigns *label* as the menu label for the next undoable operation on document *doc*. If *doc* is not specified, the value of `current_doc` is used.

If this function is called multiple times during a single undoable operation, only the label from the first call is used. Subsequent calls are ignored. The function returns a one (1) if the label will be used, or zero (0) if the call is ignored. To clear a label set by a previous call, see the `undo_menu_description_clear` on page 578 function.

An example follows:

```
local undoMsg = undo_menu_description("Great Operation", doc);
...
local ret = great_operation_action(...);
# If the above failed so doc was not modified, then clear the
# undo label (if not, it will be applied to the next operation)
if ((! ret) && undoMsg) {
undo_menu_description_clear(doc);
}
```

undo_menu_description_clear

`[0|1] =` **`undo_menu_description_clear`**`([`*`doc`*`])`

This function clears the menu label for the next undoable operation on document *doc*. If *doc* is not specified, the value of `current_doc` is used. This function should only be called to clear a label set by a previous call to `undo_menu_description` on page 578.

This function returns 1 if the undo menu label was cleared.

An example follows:

```
local undoMsg = undo_menu_description("Great Operation", doc);
...
local ret = great_operation_action(...);
# If the above failed so doc was not modified, then clear the
# undo label (if not, it will be applied to the next operation)
if ((! ret) && undoMsg) {
undo_menu_description_clear(doc);
}
```

unicode_to_entity

`name = unicode_to_entity(unicodenumber[, doc])`

Provided the Unicode character number *unicodenumber*, this function returns the equivalent character entity name. For example, if you were to give *unicodenumber* the value of `920`, the function would return the value `'Theta'`.

If the *doc* parameter is not provided, the current document will be used.

If there is no character entity name for the unicode character number *unicodenumber*, the function returns an empty string.

Note that the returned character entity names are case-sensitive and depend on the document type you are using. Use the *Arbortext-path*`/lib/charent.cf` file a reference to character entity names.

universal_file_name

`string = universal_file_name(path)`

This function returns the universal naming convention (UNC) name for the drive-based network path name *path*. The *path* parameter should refer to a remote file system that has been mounted as a local drive or a drive mapped using the Windows `subst` command. If *path* is not a network path, the absolute file name, possibly drive-based, is returned.

Example

```
$ret = universal_file_name('f:\users\george')
```

unpack

`count = unpack (template, expr[, arr])`

This function does the reverse of `pack` — it unpacks the binary data structure *expr* into the array *arr*, returning the number of elements stored in the array. If *arr* is omitted, then `unpack` extracts a single value and returns it instead.

This is similar to the Perl function of the same name. *template* has the same format as `pack` and is a sequence of characters that specify the type of each value, as follows:

- `a` — an ASCII string, unstripped
- `A` — an ASCII string, with trailing nulls and spaces removed
- `c` — a signed 8–bit character value
- `C` — an unsigned 8–bit character value
- `d` — a double-precision floating point number in native format
- `f` — a single-precision floating point number in native format
- `i` — a signed integer (32–bit) value
- `I` — an unsigned integer (32–bit) value
- `l` — a signed long value
- `L` — an unsigned long value
- `n` — a short integer value in network (big-endian) order
- `N` — a long (32–bit) value in network (big-endian) order
- `p` — a pointer to a null terminated byte string
- `P` — a pointer to a null terminated Unicode string
- `s` — a signed short (16–bit) value
- `S` — an unsigned short (16–bit) value
- `x` — skip forward a byte
- `X` — back up a byte
- `z` — a Unicode string in big-endian (network) order, null padded
- `Z` — a Unicode string in little-endian order, null padded
- `@` — go to absolute position for next field

Each character may be followed by a number that specifies a repeat count. The character and repeat count comprise a field specifier. Field specifiers may be separated by white space. For all type specifiers except, `a`, `A`, `x`, `X`, and `@`, the repeat count specifies how many values from the list of arguments are to be used. If the repeat count is `*`, then all remaining values are used.

If you specify `*` as the first character of the *template*, you are indicating that the second argument is a pointer to a buffer to unpack, and not the buffer itself. Normally, this is not needed for ACL strings but the `dl_call` function could return such a pointer. If a number follows the leading `*`, it is the size of the buffer. If the size is omitted, the `unpack` function will not detect if the template exceeds the bounds of the buffer. This feature should be used with caution since an incorrect *template* or invalid input could cause Arbortext Editor to terminate unexpectedly.

For specifiers `a` and `A`, only a single value is used. The repeat count specifies the size of the field in the input data structure and is also the size of the output string for type `a`. If type `A` was specified, then trailing nulls and spaces are stripped from the output string. The type specifiers `x`, `X`, and `@` do not use up any values. The repeat count for `x` and `X` specify how many bytes can be skipped in the input string from the current position before unpacking the next field. The field specified `@*` tells to move to the end of the string. For example:

```
unpack("@*X3a3", "abcxyz")
```

returns the last three bytes of the string, “xyz.”

Since Arbortext Editor does not support floating point as a basic type, the unpacked values corresponding to the `f` and `d` type specifiers are returned as strings. For example:

```
unpack("d", pack("d", "72.27")))
```

returns “72.

The `a` and `A` designators convert a possibly multi-byte string in the system character set to a Unicode string. Note, if a field length is specified, for example, `a7`, not all bytes may be converted to Unicode if the field ends in the middle of a multi-byte character.

The `c` and `C` designators unpack 8-bit characters, not Unicode characters.

The `p` designator points to a null terminated byte string. The resulting string variable could then be converted by calling `unpack` or `mbstoucs`.

The `z` and `Z` designators create a byte string of Unicode characters, since strings are normally 8-bit Latin1 (ISO 8859–1) characters.

Here is an example of how to byte-swap a file containing Unicode characters:

```
while ((len = read(inf, buf, 512)) > 0) {
 ustr = unpack("z*", buf);
 write(outf, pack("Z*", ustr));
}
```

This works whether the original file was in big-endian or little-endian order.

If a function called by `dl_call` returns a pointer to a string, the ACL code must convert this to an ACL string using `unpack`. A typical method would be `unpack("p", result)`. This will work but it is much better to use `unpack("*a*", result)`. Using “p” will simply widen the 8 bit string to 16 bits without doing any multi-byte to Unicode conversion. Consequently, this method does not work if the string contains any multibyte characters. Using “*a*” will convert the result string from a multibyte string to Unicode using the current locale. In both cases you are assuming that the result is an 8-bit string. If it is a Unicode string you can use unpack with “P”, “*z*”, or “*Z*”. In all cases the ACL code must know whether the result is an 8-bit string or a Unicode string.

Caution

Before using “*a*”, “*z*”, or “*Z*” to unpack the result, make sure the pointer is a valid pointer and not a null pointer.

The Perl type specifiers `%`, `b`, `B`, `h`, `H`, and `u` are currently not supported.

uri_resolve

`string = uri_resolve (uri[, catalogpath])`

This function performs a catalog lookup to resolve the URI specified by *uri*, returning the system path name for the resource. If the identifier is not found in any of the `catalog` files in the search path, the `uri_resolve` function returns a null string.

catalogpath specifies a list of directories to search for the `catalog` file. If it is not specified, the path list specified in the `set catalogpath` option, which is initialized from the environment variable *APTCATPATH*, is searched.

The `uri_resolve(uri, catpath)` function is equivalent to `catalog_resolve("URI", uri, "", "", catpath)`.

url_decode

`string = url_decode (str[, charset])`

This function decodes the URL-encoded string *str* as encoded by `url_encode` on page 583 returning its original value. If the optional *charset* is supplied and the given string contains any URL encoded multi-byte character sequences, then those sequences are decoded back into their original Unicode characters.

The following *charset* values are supported:

- iso8859-1
- iso8859-2
- iso8859-3
- iso8859-4
- iso8859-5
- iso8859-7
- iso8859-8
- iso8859-9
- ps_ascii

- ps_symbol
- jeuc
- sjis
- big5hku
- gb2312
- koi8
- ksc5601
- utf-8
- Unicode

For example, the following ACL call:

```
url_decode("A%CE%A9B", "utf-8")
```

returns the following string:

```
AΩB
```

The `%CE%A9` sequence is interpreted as a multi-byte character sequence in UTF-8. It was decoded into the Greek Omega symbol (Ω), which is Unicode character 937.

url_encode

`string = **url_encode** (*str*[, *charset*])`

This function returns the string *str* converted to into the MIME format called "`x-www-form-urlencoded`". Most non-alphanumeric characters are converted into the 3-character string "`%`*XX*", where *XX* is the two-digit hexadecimal representation of the 8–bit character. The following punctuation characters are not encoded: "-_.*".

If any Unicode character in *str* is greater than 255, then a null string is returned unless you also supply the optional *charset* (character set) parameter. If a *charset* parameter is supplied, such characters are encoded in a multi-byte sequence using the given character set and then the resulting sequence will be URL encoded.

The following *charset* values are supported:

- iso8859-1
- iso8859-2
- iso8859-3
- iso8859-4
- iso8859-5
- iso8859-7
- iso8859-8

- iso8859-9
- ps_ascii
- ps_symbol
- jeuc
- sjis
- big5hku
- gb2312
- koi8
- ksc5601
- utf-8
- Unicode

For example, Unicode character 937 is a Greek Omega symbol (Ω). The following ACL call:

```
url_encode("A" . chr(937) . "B", "utf-8")
```

returns the following string:

```
A%CE%A9B
```

The UTF-8 encoding of Unicode character 937 is a multi-byte sequence consisting of: `0xCE` and `0xA9`.

user_tag_names

`count = **user_tag_names**(*arr*[, *doc*])`

This function fills the array *arr* with a list of user-defined tags for the current document type, returning the number of tags. The first tag is stored at index 1. All previous elements in *arr* are discarded. The *doc* argument specifies the identifier of the document tree to query. If omitted or `0`, the current document is used.

username

`string = **username** ()`

This function returns the login name for the current user or a null string if no user name can be found.

validate_against_schematron

validate_against_schematron(*sch_out*[, *doc*[, *sch_file*[, *output_file*[, *phase*[, *xpath_node*]]]]])

This function is used to validate the document instance *doc* against a given list of Schematron files, defined by *sch_file*. If *doc* is not supplied or is `0`, then the current document is used. If *sch_file* is not supplied, then the Schematron file for the instance's document type is used. Each error reported by the Schematron is filled into an entry of the array *sch_out*. The array content is a string that contains both an XPath expression to the element in error and the error description from the Schematron file, separated by the `|` character. If *output_file* is specified and not empty, then the XML file with the Schematron results is saved to that file. If *phase* is supplied and not empty, then that value is passed as the phase parameter to the Schematron. If *xpath_node* is supplied and not empty, then the Schematron is only applied to the XPath node expressed by that variable.

For example, the following ACL call will validate the current document against the associated Schematron files `axdocbook1.sch` and `axdocbook2.sch`:

```
validate_against_schematron($S, 0, "axdocbook1.sch;axdocbook2.sch");
```

Note that the semi colon (`;`) is the required delimiter.

If the document or Schematron file is not valid, then the function returns `-1`. If any errors occur during the schema validation, then the function returns `-1`. Otherwise, the function returns the number of errors found, which is also the number of entries put into the *sch_out* array.

varsub

`string=`**`varsub`** `(`*`string`*`)`

This function substitutes variable names within a *string*. This function is quite useful for delaying the substitution of variables until after a call to `amo_text` or `agettext` functions.

Example

> **Note**
>
> This example uses `agettext` to retrieve a message from the default message file. Use `amo_text` to retrieve a localized message from a user-defined message file.

You need to display the following message from the default message file.

Error: No file named *$name* exists.

The problem is that the message includes a variable, *$name*, containing the file name `test.sgm`. The message is listed in the catalog with the variable name in place. Let's assume you execute `agettext` as follows:

```
msg_string = agettext \
("Error: No file named $name exists.")
```

The actual string passed to `agettext` would include the substituted value for the *$name* variable:

```
Error: No file named test.sgm exists.
```

This does not match the listing in the catalog, so the `agettext` would fail. Executing `agettext` within `varsub` solves this problem (note the use of single quotes).

```
msg_string = varsub(agettext \
('Error: No file named $name exists.')
```

In the sample above, the actual string passed to `agettext` would be:

```
Error: No file named $name exists.
```

This entry matches the entry in the default message file, so the returned message (variable name in place) is passed to `varsub`, which performs the variable substitution. The final result, stored in the variable *msg_string* would be:

```
Error: No file named test.sgm exists.
```

If the *string* includes a variable name that you do not want substituted, include an extra dollar sign (`$`) for the non-substituted variable. For example:

```
msg_string = varsub('The $$name variable is equal to $name.')
```

In this example, the result stored in *msg_string* would be:

```
The $name variable is equal to test.sgm.
```

vbscript

`string = **vbscript** (*expr*[, *window*])`

This function sends the string *expr* to the Microsoft Windows VBScript interpreter to be evaluated, returning the result as a string. If the *window* parameter is not specified, the VBScript expression is evaluated in the global VBScript script context so previously defined VBScript functions are accessible.

The optional *window* parameter is the identifier of a XUI dialog box that has a script context. If *window* is provided, the script executes in that dialog box's script context.

Example

```
path = vbscript('Application.ActiveDocument.DocumentURI')
```

Note

In the Microsoft Script Engine interfaces, "True" is represented by minus one (`1`) and "False" by zero (`0`).

window_activate

window_activate ([*window*])

This function brings to keyboard focus (activates) the window identified by *window* and sets its document to the current document. If *window* is omitted or `0`, the window associated with the current document is used.

window_add_recent_documents

window_add_recent_documents(*pathname*)

This function adds the specified document to the Windows **Start⇒Documents** menu. If *pathname* is a null string, the **Start⇒Documents** menu is cleared.

window_cascade_all

window_cascade_all ([*topwindow*])

This function cascades all Arbortext Editor windows leaving *topwindow* at the top of the cascade. If *topwindow* is omitted or `0`, the current window is used.

This function does not cascade non-Arbortext Editor windows.

window_class

window_class ([*window*])

This function returns the class name (for example, `edit`, `cmd`, `helpwin2`) for the window identified by *window*, which must be valid window identifier. Window identifiers are returned by several ACL functions, including `window_create`, `doc_window`, and `window_id`. If *window* is omitted or is `0`, the window associated with the current document is used.

window_close

window_close ([*window*])

This function iconizes the window identified by *window*. If omitted or `0`, the window associated with the current document is used.

window_count

window_count ([*class*[, *firstonly*]])

This function returns the number of existing windows of *class*, or all windows if *class* is not specified. If the argument *firstonly* is specified and non-zero, then `window_count` counts only the first window of each frame. For example, `window_count("edit",1)` would return the number of edit-class frames while `window_count("edit")` would include all edit windows, including those in split frames.

window_create

`window_create` (*`class`*`[, `*`flags`*`[, `*`doc`*`[, `*`geom`*`[, `*`parent`*`[, `*`xuipath`*`]]]]]`)

This function creates a window of the specified *class* with optional components given by *flags* and returns a unique identifier that may be used in subsequent calls to the `window_`*xxx* functions. The window created is not initially displayed. Use the `window_show` function to make the window appear.

Windows are of two types:

- Document class windows that display a document tree and have a *class* of `edit`, `helpwin[1-4]`, or `msgwin[1-4]`.
- Dialog class windows that display either a list selection dialog box of *class* `list` or `xui`.

The class determines the default geometry and, for classes other than `list`, the class-specific keymap. By default, a new keymap is created for the window on the first `map` command processed for the window. This keymap has the name `window`*n*, where *n* is the identifier of the window, and is deleted when the window is destroyed. If a `set keymap=user` command is executed for the window, or if bit 0x00040 is specified in *flags*, the global keymap for the class will be used. See the *flags* bit descriptions below.

The *flags* parameter is a bit mask that depends on the class and is defined below:

The following are the flag bits for `list` and `xui` class windows.

- `0x1` — Supply vertical scrollbar.
- `0x2` — Verify input (that is, set verify Item attribute).
- `0x4` — Supply an **Apply** button,
- `0x8` — Supply a **Help** button.
- `0x10` — For XUI dialog boxes, delete the document after the window is destroyed. In the following example, `$doc` will be destroyed automatically after `$win` is destroyed:

```
$doc = doc_open('c:\myproject\myxuifile', 1);
```

```
$win = window_create('xui', 0x10, $doc);
```

- `0x040000` — Make the new window a top-level window with its parent being the desktop. This flag is only useful to the `xui` and `list` class windows; the windows of all other window classes are created as top-level windows.

OK and **Cancel** buttons are always supplied.

The following are the flag bits for document class (all non-`list`) windows. The "(pane)" note indicates a flag that, when returned by `window_mask` on page 595, reflects the status for the pane containing the cursor (that is, last active window).

- `0x00001` — Supply vertical scrollbar (pane).
- `0x00002` — Supply menu bar.
- `0x00004` — Supply command subwindow if *class* is `edit`.
- `0x00008` — Supply message footer subwindow.
- `0x00010` — Automatically call `doc_close` on the attached document when the window is destroyed. (pane).
- `0x00020` — Supply edit toolbar (that is, Toolbar 1) (pane).
- `0x00040` — Attach the global window class keymap to the window instead of creating a private keymap (pane).
- `0x00080` — Supply horizontal scrollbar (pane).
- `0x00100` — Do `edit` command initializations, include reading the document type instance command files on page 48 (`instance.acl` and `instance.js`) and document command files on page 48 (*`docname`*`.acl` and *`docname`*`.js`) if they exist, and calling `editfilehook` on page 946 when a document is attached to the window. This bit applies only to edit class windows (pane).
- `0x00200` — Split the window into two side–by–side (that is, next to one another other) panes.

> **Note**
>
> You should not use this flag with `window_create` to create a multi-paned window. It is included in the bit mask for status checking using the `window_mask` on page 595 function. Create the window with a single flag, but then use the `window_split` on page 604 function to create multiple panes.

- `0x00400` — Split the window into two top/bottom (that is, one over the other) panes.

> **Note**
>
> You should not use this flag with `window_create` to create a multi-paned window. It is included in the bit mask for status checking using the `window_mask` function. Create the window with a single flag, but then use the `window_split` on page 604 function to create multiple panes.

- `0x00800` — Make the window a typical user edit window (as opposed to a display window).
- `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).
- `0x080000` — Do not update the `filelist` option, the Arbortext Editor **File** menu list of recently edited documents, or the Microsoft Windows list of recently edited documents with the path name associated with the *doc* parameter (if the `0x00800` flag is specified). Note that this does not apply to documents subsequently loaded in the window.

If a menu bar is requested, it must be initialized using the `menu_load` or `menu_add` commands before the window is first displayed.

If a message footer is created, error messages and output from the `message` command are displayed in the left part of the footer if the message is short enough (otherwise a dialog box is used). The `message` attribute of the `window_set` function may also be used to set the message text. Any messages directed to the message footer are considered transient and are erased on the next key or button event received in the window.

The *doc* parameter specifies the identifier of the document tree to be attached to the window. The document must not already be displayed in another window. If it is, the function returns `-1`. If *doc* is `-1`, a scratch document is created that will automatically be destroyed when the window is destroyed. In this case, the associated document type is `ascii` for `edit` class windows or the built-in `help` document type for other classes.

If the *flags* argument specifies bit `16`, the `doc_close` is called on *doc* when the window is destroyed by `window_destroy`. In this case, the normal `quit/exit` processing is done. If there are unsaved changes, and if an `exit` command was used to dismiss the window, changes are saved without prompting. If a `quit` command caused the call to `window_destroy`, you are prompted to save any unsaved changes. The *doc* parameter does not apply to `list` class windows and is ignored if given.

geom specifies the initial geometry for the window and is a string of the form WxH+X+Y, where W and H are the width and height of the window in pixels, and X and Y give the location of the upper left corner of the window. Refer to the description of the *geometry* attribute of `window_set` on page 597 for details.

`window_set` may be used to set various attributes for the window such as its title, status bar message, and destroy callback.

`window_create` returns `-1` if the window cannot be created.

parent it is an optional parameter used as the parent window for the new window. Only supports dialog class. If parent is `0`, then the active window is used.

xuipath is an optional parameter used only by edit windows 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.

window_cur_table

`gridId = `**`window_cur_table`**`(`*`[window]`*`)`

This function returns the ID of the active grid in window *window* (or in the active window, if *window* is omitted).

window_destroy

`window_destroy` `(`*`window`*`)`

This function destroys the window identified by *window*, releasing all resources allocated. If a scratch document was allocated by `window_create`, that is, no document identifier was passed to `window_create`, then it will be freed also. If a `destroyCallback` is set on the window, it will be called before the window is actually destroyed. If you execute this function on the only remaining window, Arbortext Editor will exit.

window_doc

`window_doc` `([`*`window`*`])`

This function returns the document identifier for the document tree displayed in the window specified by *window*. The *window* argument is a window identifier as returned by `window_create` or is one of the window classes such as `edit` or `cmd`. In the latter case, the window of the specified class last having focus is used. If no argument is given or is `0`, the current window is used.

window_empty

window_empty ([*window*])

This function returns 1 (True) if the window identified by *window* contains an empty scratch document.

window_enable

window_enable (*window*[, *enable*])

This function can be used to enable or disable a window. It can also be used to query the current enabled status of a window.

- *win* — A valid window identifier.
- *enable* — One of the following values:
  - `1` — Enable the given window.
  - `0` — Disable the given window.
  - `-1` — Retrieve the current status without changing the window's current enabled status.

A disabled window cannot receive input from the user. The return value is the previous enabled state of the window:

- `0` — The window was disabled.
- non-zero — The window was enabled.

window_get

window_get (*window*, *attr*[, *arr*])

This function returns the value of the attribute *attr* for the window identified by *window*. If the supplied *attr* is a `set` option, the function returns the window or view scope value of the option. If *attr* is a `set` option with a document local scope or no local scope at all, it returns nothing.

In all but one case, the attribute value is the return value of the function and the third argument *arr* should not be given. If *attr* is "items", then *arr* specifies the name of an array which is filled with a list of all items displayed in the list selection dialog box identified by *window*. In this case, the function returns the number of items stored in *arr*. See `window_set` on page 597 for a list of all available window attributes.

Examples

```
window_get(w, "geometry")
window_get(listw, "items", arr)
window_get(w, "destroyCallback")
```

window_get_columnview

window_get_columnview(*window*, *colnamesArr*, *colwidthArr*)

This function enables you to retrieve the Column view settings for a window view.

The *window* parameter is the window view for which Column view settings are being retrieved.

The *colnamesArr* parameter is an array of the column names for all columns defined in the `.dcf` file for the document type. The order of the entries in the array represents the order in which the columns are currently displayed. The first or **Outline** column is not included in the array, since it cannot be reordered and it never scrolls.

The *colwidthArr* parameter is an array parallel to *colnamesArr* that indicates the column width or whether each column is being displayed. Positive values indicate that the column is being displayed and gives the width of the column in pixels. Negative values indicate that the column is not being displayed and the default width of the column if it were showing.

The special column name, `*ScrollBoundary*`, is used to position the scroll boundary. Columns that come before the scroll boundary do not scroll horizontally and columns that come after the scroll boundary do scroll. Only one scroll boundary can be present. If a scroll boundary is not present, then only the **Outline** column is not scrolled.

The function returns the number of columns in the arrays. A zero indicates that Column view is not defined for this window. A `-1` is returned for other errors.

window_id

window_id ([*class_or_index_or_title*])

This function returns different values in the following situations:

- If *class_or_index_or_title* is a number *n*, `window_id` returns the identifier of the *n*th subwindow in the main window associated with the current document. For example, if the current document is displayed in an `edit` class window, `window_id(1)` would return the identifier of the `edit` subwindow and `window_id(2)` would return the associated command subwindow identifier.
- If *class_or_index_or_title* is one of the classes `helpwin1` through `helpwin4`, or `msgwin1` through `msgwin4`, `window_id` returns the window identifier of the corresponding predefined window, or `-1` if it does not exist. `window_id` never returns a window of one of these classes created by `window_create`, only built-in windows.
- If *class_or_index_or_title* is `edit` or `cmd`, `window_id` returns the identifier of the window of the specified class that last had focus. If the current

document is associated with a window of one of these classes, `window_id` returns that window.

class_or_index_or_title may also be one of the following predefined modeless dialog class names:

Class	Dialog Box
bookmark	Bookmarks
detail	Collapse/Expand Divisions
fileent	File Entities
find	Find/Replace
findent	Find Text or File Entity
findms	Find Marked Section
findpi	Find Processing Instruction
findtag	Find Tag/Attribute
graphent	Graphic Entities
modattrs	Modify Attributes
msp	Marked Section Parameters
notation	Notations
pgsetuped	Page Setup Editor View
spell	Spelling
styler	Arbortext Styler
tag	Insert Markup
tagtmpl	Tag Templates
textent	Text Entities

class_or_index_or_title may also specify a string to match either the title of a window or the name of a document open in a window. For example, `window_id("Document1")` returns the window identifier of the window displaying the document named “Document1”. If the string does not match a document name, the string is matched against the titles of all open windows. For example, `window_id(window_get(0,"title"))` would return the id of the current window, the same as `window_id(0)`.

window_list

`window_list` (*`arr`*`[,` *`class`*`[,` *`window`*`[,` *`flags`*`]]])`

This function fills the array *arr* with a list of all existing windows of class *class*, or all windows if *class* is not specified. It returns the number of window identifiers stored in the array. The window identifiers are returned in no particular order.

If *window* is specified, only the subwindows belonging to the frame containing *window* are returned. If the *flags* parameter is given, *window* may be given as `-1` to mean unspecified.

The optional *flags* parameter is a bitmask that limits the window ids returned. The value is constructed using `OR` with the following flags:

- `0x01` — Return only the active window in a given frame. For example, if the frame has a window split or command window, return only the window which last had keyboard focus.
- `0x02` — Return only top level windows. That is, return only those windows that do not have a parent window.

Example:

```
window_list(arr, "edit", win)
```

The function call above would return all the edit panes associated with the window specified by `win`.

window_load_component_file

`window_load_component_file``(`*`window`*`, `*`xuifile`*`)`

`window_load_component_file` adds toolbars and menus to an existing window based on the XUI definition in the specified file. Existing toolbars and menus will not be affected by the addition.

- *window* — The value of the *id* attribute of the window to be changed. This window can be an edit window or a dialog box.
- *xuifile* — The XML file defining the XUI controls to be added to the window.

Refer to the *Working with XUI (XML-based User Interface) dialog boxes* section of the *Customizer's Guide* for detailed information on XUI dialog boxes.

window_lower

`window_lower` `([`*`window`*`])`

This function moves the window identified by *window* to the bottom of the stacking order, that is, below all other windows. If *window* is omitted or `0`, the window associated with the current document is used.

window_mask

`window_mask` `([`*`window`*`])`

This function returns the flags bit mask used to create the window identified by *window*, that is, the second argument to `window_create`. This mask can be inspected to see if the window has a scrollbar, menu bar, command window, and or message footer. If *window* is omitted or `0`, the window associated with the current document is used.

A listing of the bits in the bit mask appears in the description of the `window_create` function.

Note

Some of the flags on the bit mask are specific to the different panes within a window (for example, Document Map vs edit view). Other bits can change dynamically by moving the cursor within the edit view (for example, in/out of a table if the **Auto-hide Table Toolbar** option is enabled). The pane-specific portions of the bit mask are always dictated by the pane containing the cursor (that is, last active pane).

window_name

`window_name` `([`*`window`*`])`

This function converts the window identifier *window* into a character string that can be used on commands like `caret` and `map`, for example, "window14". If *window* is omitted or `0`, the window associated with the current document is used.

Examples

```
local win_name = window_name(win_id)
caret $win_name
```

window_open

`window_open` `([`*`window`*`])`

This function un-iconizes the window identified by *window*. If the window is already visible or is not mapped, this function does nothing. If *window* is omitted or `0`, the window associated with the current document is used.

window_raise

`window_raise` `([`*`window`*`])`

This function moves the window identified by *window* to the top of the stacking order, that is, on top of all other windows. If *window* is omitted or `0`, the window associated with the current document is used.

window_redisplay

window_redisplay ([*window*[, *flags*]])

Redraws the given window. The redrawing takes effect immediately. This is useful in command scripts to show an intermediate state. By default, changes to windows made inside command scripts do not show up until the script completes.

- *window* — The window to redraw.
- *flags* — Redraw options. The following bit values are supported in the *flags* parameter.
  - 0x0001 — Center the line containing the cursor.

window_remove_split

window_remove_split (*window*[, *flag*])

This function removes one or more split windows from the frame containing the window identified by *window*. Note, that *window* is not removed.

flag specifies which split to remove and is one of the following:

- 1 — remove the vertical split
- 2 — remove the horizontal split
- 3 — remove both the horizontal and vertical splits.

If *flag* is omitted or 0, then 3 is used, that is, both splits are removed.

window_remove_split returns the number of split windows removed.

window_reset_configuration

window_reset_configuration([*win*])

This function resets all of the display options to their default settings for the window specified by *win*. If *win* is omitted or 0, the current window is used.

See the set savewindowconfiguration on page 882 command option for the list of options reset by this function.

window_set

window_set (*window*, *attr*, *value*, ...)

This function sets the values of the specified attributes for the specified window. Several attributes are generic and apply to all window classes. Other attributes are specific to a particular window class, for example, edit and list.

If *attr* is a `set` option, this function sets the window or view scope value of the option. If *attr* is a `set` option with a document local scope or no local scope, no value is set.

Window Class Attributes

Name	Value Type
background	color
canvasbackground	color
canvasforeground	color
caretMovedCallback	function name
destroyCallback	function name
focusCallback	function name
foreground	color
geometry	string
menuPostCallback	function name
message	string
parentWindow	window ID
quitCallback	function name
title	string

Edit Class Only Attributes

Name	Value Type
prompt	string

List Class Windows Only Attributes

Name	Value Type
appendItem	string
applyLabel	string
callback	function name
cancelLabel	string
default	string
deleteItem	string
embedded	0 or 1
helpLabel	string
items	array name or 0
label	string

List Class Windows Only Attributes (continued)

Name	Value Type
okLabel	string
selection	string
verifyItem	boolean

Description of attributes:

`appendItem` *string* — Appends a new item with the specified value to the list displayed in the specified list class window.

`applyLabel` *string* — Sets the label on the **APPLY** button of the list class window to the specified string.

`background` *color* — Sets the background color for the entire dialog box specified by *window*. *color* is either a named color or an RGB specification preceded by `#`.

`callback` *`func(winbutselnpos)`* — Specifies the name of a function to be called when a button is pressed in the specified `list` class window. The function is called with four arguments:

- *win* — The identifier of the list window
- *but* — A code indicating which button was pressed:
  - `0` — Cancel
  - `1` — OK
  - `2` — Apply
  - `3` — Help
  - `-1` — The window was dismissed using the window manager.
- *seln* — A string giving the current selection.
- *pos* — The ordinal number of the selection within the list or `0` if the selection is not a element of the list (`verifyItem` must be `0` in this case).

The callback function should act on the button as follows:

- `-1` — Destroy the window using `window_destroy`.
- `0` — Dismiss the window using `window_show(win, 0)` or destroy it with `window_destroy`.
- `1` — Act on the selection and dismiss or destroy the window.
- `2` — Act on the selection and leave the window up.
- `3` — Display a help message only.

This recommended behavior assumes the default button labels.

`canvasbackground` *color* — Sets the background color for the edit canvas specified by *window*. *color* is either a named color or an RGB specification preceded by `#`.

`canvasforeground` *color* — Sets the foreground color for the edit canvas specified by *window*. *color* is either a named color or an RGB specification preceded by `#`. Setting the foreground color is only effective when editing in ASCII mode. `canvasforeground` cannot be set when editing in SGML or XML mode

`caretMovedCallback` *`func(winnewold[, oldpos])`* — Specifies the name of a function to be called when the cursor is moved such that either the containing OID or the OID to the right of the cursor has changed. The function is passed up to four arguments:

- *win* — The identifier of the window.
- *new* (optional) — The new OID, that is, `oid_caret`.
- *old* (optional) — The old value of `oid_caret`.
- *oldpos* (optional) — The old position of the cursor as would be returned by `oid_caret_pos(OID)`.

If a cut operation causes the cursor to be moved, the old object identifier *old* may no longer be valid. The callback function should call `oid_valid(old)` before using *old*. Since computing the old position *oldpos* could be time consuming in large flat documents, it is recommended that the *oldpos* argument not be used unless necessary, for example, if the callback function may need to restore the old cursor position, in effect, cancelling the cursor movement.

`cancelLabel` *string* — Sets the label on the **CANCEL** button of the list class window to the specified string.

`default` *string* — Sets the default selection for the `list` class window to the specified string (which must be a member of the list). Normally, this attribute should be set when the window is created, before it is displayed.

`deleteItem` *string* — Removes the first item matching the specified string from the list displayed in the specified `list` class window.

`destroyCallback` *func*(*win*) — Specifies the name of a function to be called when the window is destroyed. The function is passed a single argument, the ID of the window being destroyed.

`embedded` — Determines the type of Arbortext Editor window. This is a read-only attribute that can only be used by the `window_get` on page 592 function. Returns `0` for a full frame or normal Arbortext Editor window. Returns `1` for an embedded frame Arbortext Editor window running in an ActiveX control.

`focusCallback` *func*(*win*) — Specifies the name of a function to be called when the window obtains focus. The function will be called only when the window changes from an unfocused state to a focused state. The function is not

called when Arbortext Editor receives focus and the desired window had focus the last time Arbortext Editor had focus. The function is passed a single argument, the ID of the window receiving focus.

`geometry` *string* — Sets the geometry of the window to the specified value, which is a string of the form WxH+X+Y, where W and H are the width and height of the window in pixels, and X and Y give the location of the upper left corner of the window. The format of the string allows just the width and height and/or position to be set by omitting fields, for example:

```
window_set(w, "geometry", "500x350")
window_set(w, "geometry", "+5-5")
```

`foreground` *color* — Sets the foreground color for the entire dialog box specified by *window*. *color* is either a named color or an RGB specification preceded by `#`. Setting the foreground color will only affect those dialog box elements that inherit the window color. For example, scrollbars and toolbars will not respond to `foreground` color changes.

`helpLabel` *string* — Sets the label on the **HELP** button of the list class window to the specified string.

`items` *array* or `0` — Sets the list of items displayed in the `list` class window to the contents of the specified array. The first element is stored at index 1. If the attribute value is `0` or the null string instead of an array name, then an empty list replaces the contents of the scrolling list.

`label` *string* — Sets the label displayed at the top of the `list` class window to the specified value.

`menuPostCallback` *func*(*win*, *menu*) — Specifies the name of a function to be called when a top level menu is posted from the menu bar associated with the window. The function is passed two arguments: *win* is the identifier of the window and *menu* is the name of the menu being posted, for example, “File”. The callback function may use the `menu_change` command to disable or enable menu items, change item labels, and so on. The function may not change the menu bar using the `menu_load`, `menu_reset`, or `menu_delete` commands. The function is not called for shortcut menus.

`message` *string* — Sets the left field message footer to the specified string. The message will appear immediately after the `window_set` function is executed since a call to `window_sync` is done automatically. This makes the message footer useful for status messages as a script progresses. The message is erased automatically when the next key or button event is received in the window. If the specified window does not have a message footer, that is, `(window_mask(w) & 8) == 0` then this attribute is ignored.

`parentWindow` *windowID* — Sets the parent window to the window identified by *parenwindowID*.

`okLabel` *string* — Sets the label on the **OK** button of the `list` class window to the specified string.

option value — Sets the specified *option* to the specified *value*. The *option* must be one of the Arbortext Editor set options (see the `set` command on page 744). The specified *option* must have a local scope of either window or view. For options that take string values, enter the value. For options that take a boolean value (that is, `on` or `off`), enter either a one (`1`) or a zero (`0`).

`prompt` *string* — Sets the command window prompt (which is normally `Command:`) associated with the specified `edit` (or `cmd`) class window. If the value is a null string `""`, then the prompt is erased. This attribute is ignored if the window does not have a `cmd` subwindow.

`quitCallback` *func*(*win*, *code*) — Specifies the name of a function to be called when the window receives a **quit** event, that is, the window is dismissed from the window manager or by a key or menu mapped to the `quit` or `exit` commands. The function is then passed two arguments:

- The ID of the window being dismissed.
- A code that indicates how the window is dismissed and has one of the values:
  - `0` — Prompt about unsaved changes [quit].
  - `1` — Save without prompting [exit].
  - `2` — Don't save, don't prompt [quit ok].

If this function returns `-1`, then `quit` will be cancelled.

`selection` *string* — Sets the selection (the type-in field) of the `list` class window to the specified string.

`title` *string* — Sets the window manager title for the window to the specified string. For example:

```
window_set(w, "title", main::progname . " Messages")
```

Once a window has its title set with the `window_set` function, Arbortext Editor will no longer change the window's title on its own (as it normally does when a new document is loaded or following a SaveAs). To get Arbortext Editor to resume setting the window title, use `window_set(win, 'title', '')`. An empty title sent to the `window_set` function will not cause the window title to be blanked, but instead clears the state that says the title can only get updated through `window_set`. If a user really wants a blank title, then `window_set(win, 'title', ' ')` would work.

`verifyItem` *value* — Changes the verification mode of the `list` class window. If the value is true (non-zero), then any input typed in the selection field must be an element of the list displayed. If false (`0`), then the selection does not need to be a member of the list. Normally, this attribute should be set when the window is created, before it is displayed.

`view` *value* — Changes the current view to the indicated value. The following values are supported:

- `edit` — Changes the current view to Edit view.
- `docmap` — Changes the current view to Document Map.
- `column` — Changes the current view to Column view.

`viewmode` *value* — Changes the view in the current window based on the indicated value. The following values are supported:

- `edit` — Changes the window to Edit view.
- `docmap` — Changes the window to a split view with Document Map and Edit view positioned based on the current settings of the `docmapside` and `docmappercent` options.
- `column` — Changes the window to a split view with Column view and Edit view positioned based on the current settings of the `docmapside` and `docmappercent` options.

window_set_columnview

`window_set_columnview`(*`window`*, *`colnamesArr`*, *`colwidthArr`*)

This function enables you to set the Column view settings for a window view.

The *window* parameter is the window view for which Column view settings are being set, or zero to indicate that the current view should be used.

The *colnamesArr* parameter is an array of the column names for all columns defined in the `.dcf` file for the document type. The order of the entries in the array represents the order in which the columns should be displayed. The first or **Outline** column is not included in the array, since it cannot be reordered and it never scrolls.

The *colwidthArr* parameter is an array parallel to *colnamesArr* that indicates the column width and whether each column should be displayed. Positive values indicate that the column should be displayed and gives the width of the column in pixels. Negative values indicate that the column should not be displayed and the default width of the column if it were showing. A value of zero indicates there should be no change to the status or width of that particular column, but its order may have changed. Valid widths are between 10 and 1200 pixels. Widths outside of that range are ignored.

The special column name, `*ScrollBoundary*`, is used to position the scroll boundary. Columns that come before the scroll boundary do not scroll horizontally and columns that come after the scroll boundary do scroll. Only one scroll boundary can be present. If a scroll boundary is not present, then only the **Outline** column is not scrolled.

The function returns `1` for success and `0` for failure.

window_show

window_show (*window*, *map*)

This function changes the state of the window identified by *window*. If *map* is true (non-zero), then the window is mapped, that is, displayed. If *map* is false (zero), then the window is withdrawn, that is, no longer displayed. In this case, the window still exists and may be redisplayed by calling `window_show` with *map* `=1`, or could be destroyed with `window_destroy`.

window_split

window_split (*win*[, *horiz*[, *percent*]])

This function splits the current edit window into two panes containing the currently open document (if any). The *win* parameter defines the window to split. The *horiz* parameter is a boolean: `1` creates a top/bottom split, `0` creates a right/left split. The *percent* parameter defines the initial proportions of the split panes, with the specified percent applying to the left pane for a right/left split, or the bottom pane for a top/bottom split.

By issuing this function twice using different values for *horiz*, you can create a Arbortext Editor window with one left pane, and two (top/bottom split) right panes.

window_state

window_state (*window*)

This function returns an integer describing the current state of the window identified by *window*, which is a window identifier as returned by `window_create`, or is one of window classes `edit` or `cmd`, or is a predefined window name `msgwin[1-4]`, or `helpwin[1-4]`. The state is one of the following:

- `-1` — invalid or non-existent window
- `0` — window is unmapped (hidden or withdrawn)
- `1` — window is minimized
- `2` — window is visible (normal)

window_sync

window_sync ()

This function forces all pending WM_PAINT messages to be processed. These actions may be necessary in some lengthy scripts to update the display immediately instead of waiting until the script finishes.

window_sync_pane

window_sync_pane (*window*)

This function synchronizes the window in the other pane to the window identified by *window*. If *window* is omitted or `0`, the current window is used.

Synchronizing the window includes setting the caret in the other window to the specified window's caret and also repositioning the other window so that the lines containing the caret are aligned, if the frame is split horizontally, that is, the windows are separated by a vertical pane.

If the frame is split into more than two windows, `window_sync_pane` behaves as follows. If *window* specifies a Document Map window, then the synced pane is the edit (non-Document Map) window which most recently had focus. If *window* specifies a non-Document Map window, then the corresponding Document Map pane is used if it exists. If there are no Document Map panes in the frame containing *window*, then the pane which previously had focus is used.

The function returns 1 (True) on success or 0 (False) if nothing was done (for example, there is no other pane or the other pane has a different document).

window_sys_close

window_sys_close ([*window*])

This function activates the **Close** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_sys_keymenu

window_sys_keymenu ([*window*])

This function gives focus to the menu for the window identified by *window*. The resulting behavior is the same as if the ALT key was pressed and released. If *window* is omitted or `0`, the current window is used.

window_sys_maximize

window_sys_maximize ([*window*])

This function activates the **Maximize** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_sys_minimize

window_sys_minimize ([*window*])

This function activates the **Minimize** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_sys_move

`window_sys_move` `([`*`window`*`])`

This function activates the **Move** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_sys_restore

`window_sys_restore` `([`*`window`*`])`

This function activates the **Restore** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_sys_size

`window_sys_size` `([`*`window`*`])`

This function activates the **Size** item on the window system control menu for the window identified by *window*. If omitted or `0`, the current window is used.

window_table_left_column

`columnId =` **`window_table_left_column`**`(`*`window, gridId`*`)`

This function returns the ID of the left most visible column of grid *gridId* in window *window* (or in the active window, if *window* is omitted).

window_table_right_column

`columnId =` **`window_table_right_column`**`(`*`window, gridId`*`)`

This function returns the ID of the rightmost visible column of grid *gridId* in window *window* (or in the active window, if *window* is omitted).

window_update

`window_update` `()`

This function forces all open document view windows to be reformatted internally (does not cause view window display to change) and sets the caret and mouse cursor for each view window.

window_xid

window_xid ([*window*])

This function returns the native window system window id or handle associated with the window specified by *window*, where *window* is one of the strings returned by the `window` function or "root" to indicate the root window. If *window* is not given, then the current window is used.

windows_disabled

windows_disabled ()

This function returns `1` (true) if a modal dialog is currently displayed or the user interface is disabled, for example, by a call to the `disable_windows` function on page 266.

This function may be useful in a timer or channel callback to delay or avoid processing if the user interface is blocked.

write (Function)

write(*fid*, *buf*[, *len*])

This function writes *len* bytes from the scalar variable *buf* to the file or channel identified by *fid*, which is an `open` identifier returned by `open_connect` or `open_accept`. If *len* is omitted, all data in *buf* is written. `write` returns the number of characters written or `-1` on failure.

Unlike `put`, `write` can handle binary data (that is, null characters). See the description of `read` for an example of a function which copies a binary file using `read` and `write`.

For output to a file identifier, `write` is implemented using the standard I/O function `fwrite` so it is acceptable to mix `put` and `write` calls.

When writing to a network channel, Arbortext Editor automatically queues any data that is not immediately transferred, for example, if the write operation would block. Unlike `read`, you do not have to be concerned about asynchronous writes.

Example

```
write(sock, "GET" . path. "HTTP/1.0/r/r/r/n")
```

If writing to a binary stream, that is, a file opened with "wb" or a network channel, `write` writes Unicode characters. Thus,

```
write(fid, "abc\n")
```

would write 8 bytes. A Unicode text file normally starts with the Unicode byte-order mark U+FEFF which can be written to a binary stream as:

```
write(fid, chr(0xfeff))
```

The byte-order mark written to a binary file this way is in the native machine order so that on a little-endian machine like an Intel-based PC, the byte-order mark would actually appear as U+FFFE in the file. Since Unicode characters are also written in native order, this will identify the Unicode file as written on a little-endian machine.

write_preferences

write_preferences (*path*[, *doc*])

This function writes the current session scope values for all the preference items to the preferences file specified by *path* without updating the current window display. If no *path* is supplied, then the preferences are written to the file Arbortext Editor looks for when reading preferences.

If a document identifier is supplied in the optional *doc* parameter, the function will write the current local scope (window, document, and view) values to the preferences file specified by *path*. In this case, the function will set the session values as well. A document identifier of `-1` writes the current session scope values to the preferences file. If document identifier is omitted, then the function writes the preference scope values to the preferences file.

If the function executes successfully, it returns a one (`1`). If the *path* is invalid, the function returns a zero (`0`).

Examples

```
write_preferences("~/arbortextuser/.arbortext.wcf")
write_preferences("C:\\arbortextuser\\arbortext.wcf",-1)
```

In Windows, a double-quoted string uses a backslash to denote an escape sequence, so you need to supply two backslashes to represent the path separator.

xpath_boolean

xpath_boolean (*expr*[, *doc*])

This function evaluates an XPath expression with respect to a document and returns `1` (true) if *expr* is true, otherwise it returns `0` (false).

- *expr* — A valid XPath expression.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.

- `count(following-sibling::*)=0` is a viable alternative to `position()=last()`

- *doc* — The identifier of the document tree to use as the context node. If omitted or `0`, the current document is used.

A typical boolean expression will use one of the XPath boolean operators (=, `!=`, <, >, >=, <=). For example,

```
count(sect1)=0
```

Non-boolean results are converted to boolean using standard XPath rules:

- The numbers zero (`0`) and NaN (not a number) are converted to `false`. Other numbers are converted to `true`.
- Node set expressions returning at least one node are converted to `true`. Empty node sets are converted to `false`.
- Strings are converted to `true` if they contain at least one character (including white space). Empty strings are converted to `false`.

> **Note**
>
> Because non-empty strings are converted to `true`, the string "false" and the expression "`string(false())`" are both converted to `true`.

To handle errors, call this function from within a `catch`.

xpath_integer

`xpath_integer` (*`expr`*`[,` *`doc`*`]`)

This function evaluates an XPath expression with respect to a document. This function evaluates *expr* as a floating-point number, rounds to the nearest integer, and returns that integer.

- *expr* — A valid XPath expression that can be represented as a number.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.

    - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *doc* — The identifier of the document tree to use as the context node. If omitted or `0`, the current document is used.

Non-numeric results are converted to numbers using standard XPath rules:

- Strings are parsed as floating-point numbers.
- Boolean expressions are converted to `1` or `0`.
- Node set expressions are first converted to strings, then parsed. A run-time error occurs if the result cannot be represented as an integer.

To handle errors, call this function from within a `catch`.

xpath_nodeset

`xpath_nodeset` (*`arr`*, *`expr`*[, *`doc`*])

This function evaluates an XPath expression with respect to a document and returns the result as an array of OIDs. This function can be used to identify a group of elements to be iterated through. It can also be used for navigation, for example, by passing one of the returned OIDs to `goto_oid`.

- *arr* — An unordered array of OIDs returned by *expr*.
- *expr* — A valid XPath expression that evaluates to a node set containing only XPath nodes that have associated OIDs.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

    - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
    - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
    - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *doc* — The identifier of the document tree to use as the context node. If omitted or `0`, the current document is used.

This function fills the array *arr* with the first OID of each XPath node returned by *expr*. The return value is the number of OIDs stored in *arr*. Duplicate OIDs may be returned if more than one XPath node resolves to the same OID. `oid_null` on page 453 is returned for any XPath node that cannot be converted to an OID (for example, namespace nodes and attributes).

To handle errors, call this function from within a `catch`.

xpath_string

`xpath_string` (*`expr`*`[,` *`doc`*`]`)

This function evaluates an XPath expression with respect to a document and returns the result of *expr* converted to a string.

- *expr* — A valid XPath expression.

  Do not use expressions that contain `position()` or `last()`. They do not work in this context. Use the `preceding-sibling` or `following-sibling` axes instead to achieve the desired results. For example:

  - `count(preceding-sibling::*)=0` will test that the context node is the first child of its parent.
  - `count(preceding-sibling::para)=1` will test that the context node is the second para child of its parent.
  - `count(following-sibling::*)=0` is a viable alternative to `position()=last()`
- *doc* — The identifier of the document tree to use as the context node. If omitted or `0`, the current document is used.

Non-string results are converted to strings using standard XPath rules:

- Boolean expressions are converted to the string "`true`" or "`false`"
- Node set expressions return the string value of the first node, or an empty string if the node set is empty.
- Numbers are not specially formatted.

To handle errors, call this function from within a `catch`.

xpath_valid

`xpath_valid` (*`expr`*)

This function checks the syntax of an XPath expression without evaluating it. It does not validate namespace prefixes, element or attribute names. `xpath_valid` returns `1` if *expr* has correct syntax. If *expr* does not have correct syntax, the function returns `0` and stores an error message in the *$main:ERROR* variable.

This function has the following parameter:

- *expr* — An XPath pattern.

Examples:

```
xpath_valid("foo()")
```

Returns `0`, because `foo` is not an XPath function.

```
xpath_valid("foo/bar")
```

Returns `1`, regardless of whether `foo` and `bar` are valid element names or not.

```
xpath_valid("myns:foo/bar")
```

Returns `1`.

zip_extract

`zip_extract` (*`zippath`*, *`outdir`*[, *`flags`*])

This function expands the given zip file into the specified output directory. Whether any existing files are automatically overwritten is under control of the optional *flags* parameter. This function does not remove any existing files from the output directory, but may overwrite some of them in some cases. If you want the output directory to be empty before expanding the zip file, then you must do that before calling this function.

The *zippath* parameter is a full path to a zip file or resource. Both file system and HTTP or HTTPS paths are supported.

The *outdir* parameter is a full path to an existing file system directory. An HTTP path is not supported.

The *flags* parameter supports the following bits:

- `0x01` — Will attempt to overwrite any existing files during extraction. If not set, the function call will fail and not overwrite any pre-existing file. Note that this will not overwrite read-only files.
- `0x02` — Will ignore any directory information encoded inside the zip file and just extract all files into the output directory itself. That is, it will flatten any existing directory structure.
- `0x04` — The same as `0x01`, but will attempt to overwrite even read-only files.

The function returns non-zero on success and zero on failure. On failure, *main::ERROR* will be setup with an appropriate error message. On failure, some files might have already been extracted. Any such files will be left alone.

5

Commands

alias

`alias` *newcmd* { cmd1 | { *cmd1*; [*cmd2*; …]} | {}}

This command maps *cmd1* and any successive commands to the name *newcmd*. You can then execute the commands by typing *newcmd* at the Arbortext Editor command line, or by mapping the alias to a specific key on the keyboard.

To alias a command that has parameters (such as `substitute`), you must include the parameters when specifying the alias or supply a symbolic parameter (such as *$1*), the value of which is to be supplied when the alias is used.

If you alias *newcmd* to more than one command, the commands to which it is aliased must be enclosed in curly braces. To define the alias name so that you can make a forward reference to it, use:

```
alias newcmd {}
```

The commands in an alias must be separated by a semicolon (;) if they appear on the same line. If each command resides on a separate line, a semicolon is legal but not mandatory. For example, you might define this alias:

```
alias setit { insert_tag figure-block; \
 insert_string -sgml "<graphic>"}
```

To use the alias shown in the example above, type `setit` at the command line. Typing `setit` sequentially runs the commands defined in the “setit” alias.

Note

The curly brace ({) that opens the alias must always be on the same line as the `alias` command.

Defining an alias in a file (for example, your user startup file) and then using the `source` command with this file does not actually run the alias. At this point, the system just defines the alias for Arbortext Editor. To actually use the alias shown in the previous example, you must type `setit` at the Arbortext Editor command line.

To easily access an alias, you can map it to a key on the keyboard. For example, you would enter the following to map the **F3** key to the `setit` alias:

```
map F3 setit
```

Following are some additional examples of the `alias` command:

```
alias divide split
ali home caret top,first
ali replace sub $*
ali repl replace $*
ali mark_char {mark begin; caret 0,+1; mark;}
ali transpose {
 caret 0,-2;
 mark_char;
```

```
 dm;
 caret 0,+1;
 paste;
}
# The following commands define individual aliases
# to define each of the three varieties of dashes
alias insert_hyphen {
 insert_string -sgml "-X";
 delete_character;
}
alias insert_ndash {insert_string -sgml "&ndash;"}
alias insert_mdash {insert_string -sgml "&mdash;"}
```

appsave

`appsave` [`-noprompt`] [`-copydt`] [`-copypacks`] [`-pedata` | `-pecompare`] [`-dir` *dir*] [*instance*]

The `appsave` command saves all the document, stylesheet, and environment information necessary to reproduce a software problem. If you are using the Arbortext Publishing Engine for document publishing, the Arbortext Publishing Engine configuration information that Arbortext Editor is using will also be saved with the rest of the application information. This information helps PTC Technical Support diagnose any problems you may be having.

Included with the saved information will be a directory named `\portable_doc`. `\portable_doc` will contain all the files necessary to open a valid version of the current document. `portable_doc` will contain the following items:

- A modified copy of the current document at the time the `appsave` command is executed. If the document is a DITA map, the map will be replaced with a preliminary resolved document (a single monolithic document with no external references). If the document is not a DITA map, its entity references, XML inclusions, and content references will be resolved.
- A `\portable_doc_files` subdirectory containing images and other files referenced by the current document. References to these files in the modified current document will be updated to refer to the files in the `portable_doc_files` directory.

The `appsave` command parses the top level XSL stylesheet to gather all of the includes and imports and move them to directories under the main XSL stylesheet location. The process recurses into all imports and includes, and the *href* attributes in each `xsl:include` and `xsl:import` are modified according to new relative locations. All absolute paths are changed to relative paths and moved to directories under the main stylesheet. All includes and imports that require network access (such as `http`, `https`, and `ftp`) are left as is.

The `appsave` command has the following optional arguments:

- `-noprompt` — Suppresses the dialog boxes related to the `appsave` process.
- `-copydt` — Saves your **custom** DTD if it's installed someplace other than the *`Arbortext-path`*`\doctypes` directory or the *`Arbortext-path`*`\custom\doctypes` directory. Document types found in either of these directories are automatically included.
- `-copypacks` — Saves all information and files associated with an installed Arbortext application.
- `-dir` *dir* — Specifies the directory where the information will be saved. If no directory is specified, the default will be `appsave\`*`architecture`*`\`*`doctypename`*.
- *instance* — Specifies the document instance to use. If no instance is specified, the current document will be used.
- `-pedata` — Transmits the current document to Arbortext Publishing Engine, which will open the document and generate an application save from the server. It will include the data requested from the choices for `-copydt` and `-copypacks`. The application save will be returned to Arbortext Editor and put into the directory specified by `-dir` *dir*.

  This option is available when you are publishing documents using Arbortext Publishing Engine and you do not specify `-pecompare`.
- `-pecompare` — Reports a comparison of the local workstation's publishing configuration and the Arbortext Publishing Engine server's configuration. This option is available when you are publishing documents using Arbortext Publishing Engine and you do not specify `-pedata`.

attribute

`attribute`

This command displays the characteristics of the character or tag before the cursor in the format:

```
caret line,col
position n of m on window (p%)
character before caret is 'char'
size font shown at screen-display-size
```

The *line,col* in the `caret` line represents the position of the cursor in the window: *line* indicates the screen line number of the current position of the cursor, and *col* indicates its inset from the left margin. *n* represents the offset or character position from the beginning of the document to the left of the cursor; *m* is the total positions in the document. *p* is the percentage this position represents. *char* is the

character to the left of the cursor. *size* is the font size at which that character will be typeset, *font* is the font family to which the character belongs and, *screen-display-size* is the actual size at which the character is displayed in the Edit pane.

Example

```
att
```

autoload

`autoload` {*func() file* | *alias file*}

The `autoload` command defines the function named *func* or the alias named *alias* to be loaded from the file *file*. The name is defined so that subsequent references are valid, but the file is not read until the function or alias is executed. This permits command packages to be loaded on demand instead of being sourced at start up.

If *file* is not a path name, that is, does not contain any slashes, the list of directories given in the `loadpath` option is searched for a file named *file*. The default command file extension `.acl` is supplied if necessary.

If the function or alias name is already defined other than by an `autoload` command, the `autoload` command does nothing.

Examples

```
autoload version_panel version.acl
autoload sort() /packages/sort.acl
```

beep

`beep`

This command sounds a beep. Useful for removing a default mapping from a key or signaling an error in an alias or a function.

Example

```
beep
# This replaces the unwanted keymapping "alt+shift+p"
# with a beep; the beep indicates that the key no
# longer performs a useful function.
map alt+shift+p beep
```

break

`break`

Immediately exits the innermost enclosing `while` or `for` loop or case block of a `switch` statement.

Examples

```
break
# The following alias only works in ascii files
alias goto_line {
 caret first,first;
 caret 0,+1;
 if ($currentline == 2) {caret first,first;}
 $lnr = $1;
 $n = 1;
 while ($n < $lnr) {
 caret 0,'(^$)|(^.)' -nows -e;
 if ($status != 0) {
 execute message "File only has $n lines.";
 break;
}
 $n++;
}
 unsetvar n lnr
}
```

caret

`caret` { mouse | nextcell | prevcell | nextword | prevword | [*window* | nextwindow | prevwindow] [warp | nowarp] [line,col] [`-c` | `-noc`] [`-e` | `-noe`] [`-t` | `-not`] [`-wrapscan` | `-nowrapscan`] [`-q` | `-noq`] [`-screen` | `-noscreen`] [`-arrow` | `-noarrow`]}

The `caret` command is used to change the location of the cursor.

- `caret mouse` — Positions the cursor at the current mouse arrow location. Note that if you are mapping a key to `caret mouse`, it is often desirable to precede the `caret` command with a `clear_mark` command on page 627. Typically, it is not recommended that you use `caret` commands in scripts, as the command is affected by the screen position. The absolute form of the `caret` command is meant to be typed at the command line or mapped to a key.
- `caret nextcell` — Applies only when the cursor is in a table cell. This command moves the cursor to the end of the next cell in the current table. If the cursor is in the last cell of a column, it is moved to the first cell in the next row. If the cursor is in the last row and column of the table, it is moved to the first cell in the table.
- `caret prevcell` — Applies only when the cursor is in a table cell. This command moves the cursor to the end of the previous cell in the current table. If the cursor is in the first cell of a column, it is moved to the last cell in the

previous row. If the cursor is in the first row and column of the table, it is moved to the last cell in the table.

- `caret nextword` — Moves the cursor to the beginning of the next word.
- `caret prevword` — Moves the cursor to the beginning of the previous word.
- The *window* command variable can be any of the following:
  - `edit` (the default) designates the Edit pane
  - `cmd` designates the command line
  - `list` moves the focus to the last active list selection dialog (the *line,col* argument cannot be used with this option)
  - `helpwin` (also called `helpwin1`), `helpwin2`, `helpwin3`, `helpwin4`, `msgwin` (also called `msgwin1`), `msgwin2`, `msgwin3`, and `msgwin4` designate multiple Help windows. The `msgwin` values display Arbortext Editor messages and output from the `show`, `eval`, and other commands.
  - *win-name* moves the focus to the specified window. *win-name* is a string returned by `window_name`.

  `nextwindow` (used with keymappings) moves the cursor from window to window. `next` is a synonym for `nextwindow`.

  `prevwindow` moves the focus (and pointer) to the previous window, in the direction opposite of `nextwindow`. `prev` is a synonym for `prevwindow`.
- `warp` (the default) moves both the cursor and the pointer to the designated window. `nowarp` moves the cursor without moving the pointer. If click-to-type focus is used, then `warp` sets the keyboard focus to the specified window instead of moving the pointer.
- The `caret` command positions the cursor immediately in front of the character at the specified screen location *line,col* where *line* is an integer representing the number of lines down on the screen (this includes blank lines) and *col* is an integer representing the number of characters and spaces (that is, columns) over from the left margin.

  line and *col* values can be either absolute or relative to the current cursor position. When *line* and *col* are relative values, a plus sign (+) means move right or move down this number and a minus sign (−) means move up or left. In addition to simple integers, values for *line* can be:
  - `firstline` or `first` (the first line in the document).
  - `middleline` or `middle` (the middle of the document)
  - `lastline` or `last` (the last line in the document)

- `topline` or `top` (the first line on the screen)
- `centerline` or `center` (the middle of the screen)
- `bottomline` or `bottom` (the last line on the screen)
- `currentline` or `current` (the line the cursor is presently on)
- `beginparagraph` or `begin` (the first line of the paragraph in which the cursor is positioned)
- `endparagraph` or `end` (the last line of the paragraph in which the cursor is positioned)
- `screensize` (the number of lines on the screen)

col can also be

- `firstcolumn` or `first`
- `middlecolumn` or `middle`
- `lastcolumn` or `last`
- `currentcolumn` or `current`

Values for *line* and *col* can also be `0` (zero) or strings, specified in the same manner as for the `find` command on page 646 and `substitute` command on page 715. `0` means keep the relative pixel position for this coordinate. When a string is used, Arbortext Editor searches forward from the cursor until it finds the string, then uses the string (if found) as the value. The cursor position following the matched string is used as the location (for example, `caret 0,' '+1` moves the cursor one character past the next blank). You can also search backward using a relative location expression (for example, the command `caret 0,-'.'+2` might be used to position the cursor at the start of the current sentence).

The operators +, -, / (for division), and * (for multiplication) may also be combined with integers and with the *line* and *column* variables to form expressions. Thus, the command `caret (bottom-top)/2,0` would position the cursor on the middle line of the screen in the same horizontal position. Note how parentheses designate the order of evaluation in the above example.

- The `-screen` and `-noscreen` modifiers control whether invisible characters (for example, tags) are skipped when they aren't displayed. The `-noscreen` modifier specifies that no characters are to be skipped. The `-screen` modifier specifies that invisible characters should be skipped. If neither `-screen` or `-noscreen` are specified, the default is `-noscreen`. For an example of the `-screen` modifier, consider the following code fragment:

```
<para>Using the Caret command.</para>
```

```
<para>Fun with the -screen option!</para>
```

Assume that the cursor is between the period and the first `para` end tag, and the current display mode is **Full Tags**. If you issued the command `caret 0,+1 -screen`, the cursor would move one character to the right and appear just to the right of the first `para` end tag. However, if the current display mode is **No Tags**, the command `caret 0,+1 -screen` would skip the invisible tag characters and place the cursor just to the left of the "F" in "Fun".

- The `-arrow` and `-noarrow` modifiers are only used when `-screen` is set and the Edit window has tag display turned off. Setting `-arrow` specifies that invisible characters are not skipped (like `-noscreen`), however, invisible generated text and the contents of collapsed items are skipped (like `-screen`). If neither `-arrow` nor `-noarrow` are specified, the default `-noarrow` is assumed.
- The remaining modifiers (`c`, `noc`, `e`, `noe`, `t`, `not`, `q`, `noq`, `wrapscan`, and `nowrapscan`) work like the modifiers for the `find` command.

Examples

```
# position caret at current mouse location
car mouse
car nextcell
car prevwindow
# second line, tenth column
car 2,10
# down two lines, right ten columns
car +2,+10
# this command moves the cursor to the next paragraph
# in the edit window without moving the mouse pointer
car edit nowarp 0,/<para>/ -t
car bottom+1,first
car -2,0
car current-2,current
car bottom,last
car last,last
car begin,current
car current,/ /
car cmd 1,1
car (bottom-top)/2,0
car 0,/apples|oranges/ -e
car 0,/Doctor/ -c
```

cd

`cd` *pathname*

This command changes the current working directory to the directory specified by *pathname*. If *pathname* specifies a drive name only, then the current drive is changed.

Examples

```
cd /docs/book/chap1
cd ..
```

change_entity

`change_entity` [`-all`] *newname*

This alias replaces the entity reference before the cursor with a reference to *newname*. When `-all` is specified, all occurrences are replaced.

`ce` is a synonym for `change_entity`.

Examples

```
ce diag
ce -all ati
```

change_tag

`change_tag` [[`-all`] *newtagname* | `-panel`]

In the Edit view, this command renames the closest tag, tag pair, or entity reference to the left of the cursor to *newtagname*. In the Document Map view, this command renames the closest tag, tag pair, or entity reference to the right of the cursor if the cursor is at the start of a line (that is, between the element icon and the element name); you can control this behavior with the `set docmapcurrenttag=off` option.

Note

If you have applied an alias map to the document, *newtagname* can be either an alias or a real name.

Typing `change_tag` (or simply) at the command line brings up a panel that serves as a palette of all tags in context at the point of the tag being changed. If you have applied an alias map to your document and **Tags** is the selected **Mode**, the **Elements** list will display aliases, rather than real names, for tags that have been assigned an alias. You must direct the panel to perform an action before you can continue working. Specifying `-all` renames all occurrences of the closest tag left of the cursor to *newtagname*. If no arguments are given in the command, `-panel`is the default, which causes the **Change Tag** panel to appear.

`ct` is a synonym for `change_tag`.

Examples

```
ct
ct _comment
ct -all ii
```

check_completeness

`check_completeness` [`-full`] [output=*outspec*]

This command checks whether all the components that are required by the DTD are present. If elements are missing, it provides a listing of problems and supplies missing elements where the fix is unambiguous.

Note

If you have applied an alias map to the document, the element names in the panel are aliases.

It also checks for ID references for undeclared or multiply-defined IDs. In addition, it checks for mismatches for text IDs, such as saving text to a counter ID. By default, this command checks those parts of a document that have changed since a previous completeness check including file entities. If the `-full` option is specified, the entire document is validated including any SUBDOC file entities (SUBDOC is a rarely used SGML feature). If a document is saved and exited with its status incomplete, the next completeness check done on the document defaults to `-full`.

outspec can be any of the following:

- The name of a file (this can be a complete path name). A right angle bracket (>) preceding the file name causes the result of the `check_completeness` command to be appended to the end of the file.
- A question mark (`?`) preceding the output file name. If the file name starts with a question mark, the file name is a variable name whose value is set to the output produced by the command. If the second character is a right angle bracket (>), the output is appended to the current value of the variable instead of replacing it.
- An exclamation point (`!`) preceding an output specifier. This suppresses the normal prompt to confirm overwriting an existing file.

`cc` is a synonym for `check_completeness`.

Examples

```
cc
cc -full
```

clean_cache

`clean_cache` [-verbose] [-noexec] [-help] [-size *n*] [-time *d*] [directory]

The `clean_cache` alias can be used to manage the auxiliary publishing files stored in the Arbortext Editor cache directory. The algorithm for selecting which directories to remove is based on the time a directory has spent in the cache without being modified. The available arguments are described below:

- `-verbose` — Displays information messages while the alias runs.
- `-noexec` — Displays a listing of the directories to be removed, but does not actually remove the directories.
- `-help` — Displays command syntax.
- `-size` *n* — Reduces the size of the cache to *n* Kilobytes.
- `-time` *d* — Removes files that have been in the cache (unchanged) for *d* days or more.
- `directory` — Specify the cache directory to clean. If not specified, Arbortext Editor uses the location set by the *APTCACHE* environment variable. If *APTCACHE* is not set, Arbortext Editor uses the default location. The default directory is typically `C:\Documents and Settings\`*`username`*`\Local Settings\Application Data\PTC\Arbortext\Editor\.aptcache`.

Caution

The `clean_cache` alias permanently deletes all data in the specified directory and its subdirectories, not just Arbortext Editor cached data. Use the `-noexec` and `-verbose` arguments to ensure that `clean_cache` will delete the exact data you intend.

clear_mark

`clear_mark` [ended | begun | all | noextend]

This command deselects the currently highlighted selection. The `ended` option (the default) clears the highlighting only if the selection is ended. The `begun` option clears the highlighting only if the selection has been started, but is not yet ended. The `all` option clears the highlighting regardless of the selection status. The `noextend` option clears the selection (as opposed to extending it), even if `set extendselection = on`.

`clm` is a synonym for `clear_mark`.

Examples

```
clm
clm begun
clm all
```

comment

`comment` *text*

This command causes the command parser to ignore everything from the `comment` command to the next semicolon (;), the next right curly brace (}), or the end of the line—whichever comes first.

The pound sign (#) provides a second type of comment which suppresses everything that follows it on the line.

Comments can be used to annotate a file or to prevent commands from being executed.

`com` is a synonym for `comment`.

Examples

```
com The pscr alias prints the screen;
ali pscr print unformatted screen
# This is useful when working on documentation
# map f3 {it italic; is /Arbortext/; caret 0,+1}
alias C_ {comment};
# Uncomment message command when debugging
$d=$docname
$m=$d.".menu"; C_ message "$d/$m"; edit $d/$m
caret 0,/<.*>/; # put caret after next start tag
# The following alias redefines the "save" command
# to be nonoperative:
alias save {comment}
```

compile_doctype

`compile_doctype` [`-nowarn`] [`-noprompt`] [[`-catpath`] [*cat-path*]] ['[`-log`] [*log-path*]] [[`-dcl`] [*dcl-path*]] [[`-pubid`] [*public-id*]] [[`-toptag`] [*top-tag*]] [[*doctypeDir*] | *dtdPath*] [[`-xml`] | `-xml`]

The `compile_doctype` alias compiles the document type specified by the directory *doctypeDir* or the DTD file path *dtdPath*. If no parameters are specified, the Compile Document Type dialog box displays. This command is not supported for the compact installation of Arbortext Editor.

The `-nowarn` option ignores all warnings encountered during the compilation. By default, all errors and warnings are displayed in a message window.

The `-noprompt` option prevents display of the dialog box prompting the user to confirm all compilation parameters

The `-catpath` '*cat-path*' option allows the user to specify the catalog path to use for the compilation. The path entered must be enclosed in single quotes (for example, `'cat-path'`). If you enter multiple directories in the path, separate each directory with a semicolon (`;`). By default, the value of `option('catalogpath')` is used.

Note

You can display the current catalog path by typing `eval option(catalogpath)` on the Arbortext Editor command line.

The `-log` *log-path* option allows users to specify a file path to redirect all messages, errors, and warnings generated during document type compilation. By default, the Arbortext Editor will display messages in the message window.

The `-dcl` *dcl-path* option specifies the path to the SGML Declaration for SGML document types. If you do not specify a path, Arbortext Editor will use the appropriate default. You do not need to specify a `-dcl` *dcl-path* if you are compiling XML document types. In Arbortext Editor, XML documents are compiled using predefined SGML Declaration values for XML.

The `-pubid` '*public-id*' option specifies the public ID to use to lookup using the catalog path the SGML Declaration to use in compiling the document type. This is used only if the `-dcl` option is not provided. You must enclose the ID in single quotes (`'`) if the *public-id* option contains any white space.

The `-toptag` *top-tag* option specifies the top-level element for the document type DTD that is unwrapped, (no <!DOCTYPE wrapper). If this is not provided and the document type is one of the Arbortext Editor distributed document types (for example, DocBook), then the proper top-level tag is used.

The `-sgml|-xml` option specifies whether the document type is SGML or XML. If this option is not specified, the value of `set compilesgml` command determines the type.

continue

`continue`

This command starts the next iteration of the innermost nested command, enclosing the `while` or `for` loop.

Example

```
continue
```

copy_file

`copy_file` { *oldfile newfile* | *file1* [*file2* …] *dir*}

This command creates the file `newfile` as a copy of the file `oldfile`. The destination can be a WebDAV resource provided the target HTTP resource supports WebDAV.

In the second form, `copy_file` *file1* [*file2*] ... *dir*, each file name is copied to the specified directory which must already exist. The base name of the destination file corresponds to the original file name. If a source file name is a directory, the directory is recursively copied.

`cp` is a synonym for `copy_file`.

Examples

```
cp formlet.sgm /jjs
cp http://www.ptc.com/examples/testfile.xml /dkc
copy_file c:/temp/file.xml http://our_server/webdav/file.xml
cp c:\prj1\publ.sgm c:\prj2\arbortext
cp book.sgm book.acl /book
```

copy_keymap

`copy_keymap` *name newname*

This command creates a new keymap named *newname* as a copy of the keymap *name*. The keymap called *name* was created by `define_keymap` or `copy_keymap` commands or is a window class name such as `edit`, `cmd`, or `helpwin`. In the latter case, the new keymap will have all the user-level mappings for the specified class.

The new keymap starts out with the same mappings as *name* but subsequent `map` commands to *name* do not affect *newname* and vice-versa. If *name* is a prefix keymap, then *newname* is created as a prefix keymap.

The keymap *newname* must not be one of the predefined maps `system` or `user` or a window class name such as `edit`, `cmd`, or `helpwin`.

`ckm` is a synonym for `copy_keymap`.

Examples

```
copy_keymap edit edit2_map
ckm window7 save_map
```

copy_mark

`copy_mark` [`-append`] [`-noclm`] [*buffername*]

This command copies the currently selected text to the specified paste buffer. If a buffer name is supplied, then that buffer is used. Otherwise the current paste buffer is used. By default, the selection is cleared after the selection is copied. If `-append` is specified, text is inserted at the end of the buffer rather than completely replacing its contents. If `-noclm` is specified, the current selection is not cleared.

Change tracking markup is never copied to a buffer on a cut or copy. What is copied is determined by the `set viewchangetracking` on page 923 command. If the command is set to show `changeshighlighted`, then the `changesapplied` view of the selection is used instead.

`cpm` is a synonym for `copy_mark`.

Examples

```
cpm
cpm buf1
cpm -noclm
cpm -append buf1
```

count_file_entities

`count_file_entities`

This command will report any file entity names and how many times they appear in the current document.

Examples

```
count_file_entities
```

count_graphic_entities

`count_graphic_entities`

This command will report any graphic entity names and how many times they appear in the current document.

Examples

```
count_graphic_entities
```

count_marked_sections

`count_marked_sections`

This command will report any marked section names and how many times they appear in the current document.

Examples

```
count_marked_sections
```

count_notations

`count_notations`

This command will report any notation names and how many times they appear in the current document.

Examples
`count_notations`

count_text_entities

`count_text_entities`

This command will report any text entity names and how many times they appear in the current document.

Examples
`count_text_entities`

create_file_entity

`create_file_entity` [*name* [*sysid* [*pubid* [*type*]]]]

This alias creates a new document of the same type as the document being edited and then inserts, at the cursor, an entity reference to the new document. Any selected (highlighted) text is moved to the new document. If you simply enter `create_file_entity` with no arguments, you are prompted for an entity name and for the system and public identifiers (which are optional).

name is the file entity name, which is used to construct the reference. If you enter the name of a file entity that has already been declared and there are no conflicts with system and public identifiers for that entity, information from the existing declaration is used. If there are conflicts, you are prompted to resolve them. If the name you supply is unique, this alias constructs a file entity declaration for the new document and adds this declaration to the private markup section of the document instance being edited.

sysid is the system identifier for the file entity, which is also used as the path name for the document instance being created. If no system or public identifier is specified, the entity name is used as the file name, and the file is placed in the current working directory.

Note

If a text string, the system identifier, or the public identifier includes spaces or special characters (characters other than letters, digits, and underscores), it needs to be enclosed in quotation marks.

pubid is the public identifier for the file entity. An attempt is made to find this identifier in one of the `catalog` files in the directories identified by the `set catalogpath` option (typically the `doctypes` subdirectory of *`Arbortext-path`*). If a mapping for the identifier is found in one of these files, then the path name to which the identifier is mapped is used as path name for the new document.

Note

When both system and public identifiers are given, if a mapping for the public identifier can be found in an entity map file, the public identifier takes precedence and the system identifier is not used.

type is the type of entity. `SUBDOC` is the only supported option.

`cfe` is a synonym for `create_file_entity`.

Examples

```
cfe
cfe list-of-users
cfe users '/sys/usr-list' '-//ATI//ENTITIES User List//EN'
```

create_text_entity

`create_text_entity` [*name* [*type*]]

This alias creates a new text entity and then inserts, at the cursor, an entity reference to the new text entity. Any selected text is moved to the new text entity. If you simply enter `create_text_entity` with no arguments, you are prompted for an entity name.

name is the text entity name, which is used to construct the reference.

type is the optional type of the entity and the only supported option is `CDATA`". This only applies to SGML documents.

`cte` is a synonym for `create_text_entity`.

Examples

```
cte
```

```
cte product-name
```

declare_entity

`declare_entity` [*text* [*name* [*textstring*]] | file [*name* [*sysid* [*pubid*]]] | graphic [*name* [*notation* [*sysid* [*pubid*]]]]]

This alias creates either a text, file, or graphic entity reference, as specified, prompting for input as necessary.

For explanations of the arguments, see `declare_text_entity` on page 637, `declare_file_entity` on page 634, and `declare_graphic_entity` on page 635.

Note

If a text string, the system identifier, or the public identifier includes spaces or special characters (characters other than letters, digits, and underscores), it needs to be enclosed in quotation marks.

`de` is a synonym for `declare_entity`.

Examples

```
de
de text atext Arbortext
de file copyright "USER-PATH/project1/copyright"
```

declare_file_entity

`declare_file_entity` *name* [*sysid* [*pubid* [*type*]]]

This alias creates a file entity reference, prompting for input as necessary.

name is the file entity name, which is used to construct the reference. If you enter the name of a file entity that has already been declared and there are no conflicts with system and public identifiers for that entity, information from the existing declaration is used. If there are conflicts, you are prompted to resolve them. If the name you supply is unique, this alias constructs a file entity declaration for the new document and adds this declaration to the private markup section of the document instance being edited.

Note

If the system identifier or the public identifier includes spaces or special characters (characters other than letters, digits, and underscores), it needs to be enclosed in quotation marks.

sysid is the system identifier, or path name, for the file entity.

pubid is the public identifier for the file entity. The identifier must be mapped in one of the `catalog` files in the directories identified by the `set catalogpath` option (typically the `doctypes` subdirectory of *`Arbortext-path`*). When entered from a command file or the command line, the public identifier must be enclosed in quotation marks.

type is the type of entity. `SUBDOC` is the only supported option.

`dfe` is a synonym for `declare_file_entity`.

Examples

```
dfe
dfe ch1 '/doc/startup' '-//ATI//docarch man ch1//EN'
```

declare_graphic_entity

`declare_graphic_entity` [*entity* [*notation* [*sysid* [*pubid*]]]]

This alias creates a graphic entity reference, prompting for input as necessary.

notation is an SGML notation for the data type of the graphic.

sysid is the system identifier, or path name, for the graphic entity.

Note

If this system identifier or public identifier includes spaces or special characters (characters other than letters, digits, and underscores), the whole string needs to be enclosed in quotation marks.

pubid is the public identifier for the graphic entity. The identifier must be mapped in one of the catalog files in the directories identified by `set catalogpath` option (typically the `doctypes` directory of *`Arbortext-path`*). When entered from a command file or the command line, the public identifier must be enclosed in quotation marks.

`dge` is a synonym for `declare_graphic_entity`.

Examples

```
dge
dge turbogen tif '/ati/graphics/genr.tif'
```

declare_ms_parameter

`declare_ms_parameter` [*parametername* [*status*]]

This alias defines a marked section parameter entity name, prompting for input as necessary.

status consists of status keywords and references to other marked section parameters. Status keywords, in order of precedence, are as follows:

- `IGNORE` (abbreviated `ig`)
- `CDATA` (abbreviated `c`)
- `RCDATA` (abbreviated `r`)
- `INCLUDE` (abbreviated `in`)
- `TEMP` (abbreviated `t`)

References to other marked section parameter entities must be prefixed with a percent sign. To enter a status containing multiple keywords or any references to other marked sections, enclose the status string in quotation marks.

`dmsp` is a synonym for `declare_ms_parameter`.

Examples

```
dmsp note-to-editor ig
dmsp sgml-code CDATA
dmsp pass3 'ig t'
dmsp irish include
dmsp gaelic '%irish'
```

declare_notation

`declare_notation` [*notationname* [*sysid* [*pubid*]]]

This alias creates a notation declaration, prompting for input as necessary.

sysid is the system identifier, or path name, for the notation. If this string includes spaces or special characters (characters other than letters, digits, and underscores), the whole string needs to be enclosed in quotation marks.

pubid is the public identifier for the notation. To be used by Arbortext Editor, the identifier must be mapped in one of the `catalog` files in the directories identified by the `set catalogpath` option (typically the `doctypes` subdirectory of *`Arbortext-path`*). When entered from a command file or the command line, the public identifier must be enclosed in quotation marks.

Note

If this system identifier or public identifier includes spaces or special characters (characters other than letters, digits, and underscores), the whole string needs to be enclosed in quotation marks.

is a synonym for `declare_notation`.

Examples

```
dn
dn tex "/ati/help/tex.help"
dn cgm '/iso/cgm' 'ISO 82/4//NOTATION Clear text
 encoding//EN'
```

The last example should be entered all on one line.

declare_text_entity

`declare_text_entity` [*name* [*textstring* [*type*]]]

This alias creates a text entity reference, prompting for input as necessary.

name is the text entity name, which is used to construct the reference.

textstring is the text that replaces the entity reference in the formatted document. If this string includes spaces or special characters (characters other than letters, digits, and underscores), the whole string needs to be enclosed in quotation marks.

type is the type of the entity. `CDATA` is the only supported option.

is a synonym for `declare_text_entity`.

Examples

```
dte
dte hap HaPPSys
dte sgmled 'SGML Editor'
```

define_keymap

`define_keymap` [`-prefix`] *name*

This command creates a new keymap named *name* that initially has no keys mapped. *name* has the same syntax as variables names: it must start with a letter and is composed of letters, digits, and underscores. The keymap name must not be one of the predefined maps such as `system` or `user`, a window class name such as `edit`, `cmd`, `helpwin`, etc., or a window name as returned by `window_name` (for example, `window4`, `window7`, `window11`).

Key and mouse bindings may be assigned to the keymap using the `map` command. The keymap may be attached to the current window by using the command `set keymap=`*`name`*.

If `-prefix` is specified, then the keymap is created as a prefix keymap. A prefix keymap differs from a normal keymap in that it is attached to a window by a key press (termed a prefix key) and remains attached to the window only for the next key or button press. This allows a command or command list to be mapped to a sequence of key presses. For example, the key sequence **CTRL+X**, **CTRL+S** is mapped to the `save` command in the example that follows.

`dkm` is a synonym for `define_keymap`.

Examples

```
define_keymap -prefix ctrl_x_map
map @ctrl_x_map control-s save
define_keymap emacs
map @emacs control-x set keymap=ctrl_x_map
```

define_tag

`define_tag` [*newtagname oldtagname*]

This command creates the user-defined tag *newtagname* as an alias of the tag *oldtagname*. *newtagname* can be made up of any combination of letters, numbers, dashes (-), underscores (_), and periods (.). If tag names are not specified, it brings up the **Create User-defined Tag** dialog box, which prompts for this input.

Note

If you have applied an alias map to the document, *oldtagname* can be an alias or a real name.

`dft` is a synonym for `define_tag`.

Examples

```
dft invisibleindexterm indexterm
dft editor comment
```

delete_buffer

`delete_buffer` [`-all` | *buffername*]

This command empties and deletes the specified paste buffer. If `-all` is specified, it empties and deletes all buffers, including the default buffer. If no buffername is specified, it deletes the contents of the default buffer.

`db` is a synonym for `delete_buffer`.

Examples

```
db
db -all
db buf1
```

delete_character

`delete_character` [-f]

This command erases the character before the cursor. If the optional *-f* argument is set, the command erases the character after the cursor.

`dc` is a synonym for `delete_character`.

Example

```
dc
```

delete_entity

`delete_entity`

This command deletes the entity reference to the left of the cursor.

`dle` is a synonym for `delete_entity`.

Example

```
dle
```

delete_lms

`delete_lms -all`

`delete_lms -all` deletes all local modifications to the tag preceding the cursor (that is, it removes all attributes).

`dlm` is a synonym for `delete_lms`.

Examples

```
dlm
dlm -all
```

delete_mark

`delete_mark` [[`-append`] [*buffername*] | null]

This command deletes the currently selected region. If the `deletespaces` option is set to `on`, then `delete_mark` will also remove a space if the deletion would result in two consecutive spaces. The deleted region is moved to the paste buffer specified. If no buffer name is supplied, the current paste buffer is used (see `set paste=`*buffername* on page 861). If the *buffername* is `null`, the selection is

deleted without being copied to a paste buffer. If the `-append` option is selected, text is inserted at the end of the buffer rather than completely replacing its contents. This command corresponds to the **Delete** menu item on the default **Edit** menu.

`dm` is a synonym for `delete_mark`.

Examples

```
dm
dm bufA
dm -append buf2
```

delete_tag

`delete_tag`

In the Edit view, this command deletes the tag to the left of the cursor. In the Document Map view, this command deletes the tag to the right of the cursor if the cursor is at the start of a line (that is, between the element icon and the element name); you can control this behavior with the `set docmapcurrenttag=off` option.

For these purposes, tags are any of the following markup icons: element tag or tag pair, processing instruction, user-defined tag, or entity reference.

`dt` is a synonym for `delete_tag`.

Example

```
dt
```

detail

`detail` [[`-selection`] | `-caret` | `-mouse` | `-all`] [`-full` | `-none` | *n* | *tagname* | `-toggle`]

This command collapses the contents of the specified tag pair to a plus sign with surrounding brackets, [+], or expands the contents of a collapsed structure. Special handling for divisions (such as chapters) causes the division heading text to continue to be displayed when the division is collapsed.

The first set of options determines the region on which the command operates:

- `-selection` (the default) indicates the region that is currently selected (or highlighted) if there is a selection; otherwise, the region enclosed by the document tag pair (that is, the entire document). However, `-toggle` is not a valid option for `-selection`.
- `-caret` indicates the region enclosed by the first tag pair to the left of the cursor.

- `-mouse` can indicate one of two regions: the region enclosed by the tag pair to which the pointer is pointing or the region enclosed by the first tag pair to the left of the pointer.
- `-all` indicates the region enclosed by the document tag pair (that is, the entire document).

The rest of the options determine how much detail to show:

- `-full` expands all collapsed structures in the indicated region so that all detail is shown.
- `-none` collapses all structures in the indicated region so that no detail is shown.
- *n* allows you to type a number indicating the level of detail to be shown. If the number is entered without a leading + or - sign, detail is shown down to the *n* th level (relative to the beginning of the region being operated on) and collapsed below that. *-n* shows *n* less levels of detail. *+n* shows *n* more levels of detail.
- *tagname* allows you to specify the type of tag pair to be collapsed. Arbortext Editor determines the nesting level for the first occurrence of the tag *tagname*, and then collapses everything from that nesting level down.

> **Note**
>
> If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

- `-toggle` (the default) is used to alternate between two levels of detail. For `-mouse` and `-caret`, it toggles between collapsing and expanding the tag indicated. If less than full detail is currently shown and `-all` is specified, the current level of detail is remembered, and full detail is shown. If full detail is currently shown, then detail is shown to the last remembered level. `-toggle` is not a valid option for a selection.
- `-panel` displays the detail dialog box to collapse or expand divisions.

Examples

```
det
det -1
det -all
det -selection par
det -all 5
det -selection +1
det -selection -1
det -all -toggle
```

doc_flatten

`doc_flatten` { file | text | all} [undeclare]

This alias replaces the text and/or file entity references with the corresponding entity content. It always works on the entire document. The command should be used with caution, because when the document is saved, all data about the prior existence of entity references is lost. It is good practice to perform a **File ► Save As** after running the `doc_flatten` alias. The following arguments control the scope of the alias' execution:

- `file` — flattens file entities only.
- `text` — flattens text entities only.
- `both` — flattens both text and file entities.
- `include` — flattens XIncludes only.
- `all` — flattens all text and file entities as well as XIncludes.

By default, the entity declarations are retained. If the optional parameter `undeclare` is provided, then the declarations are removed.

edit

`edit` [`-newwindow` | `-ok`] [`-readonly`] [`-xml` | `-untaggedascii`] [`-encoding` *string*] [`-cc` | `-nocc`] [`-cd`] [`-stylesheet` *name*] [`-catalog` *dir*] [*filename* | `-last` | `-current`]

This command replaces the document in the Edit pane with the file specified by *filename*.

If a *filename* argument is not specified and the `-catalog` option does not specify a document entity, `edit` displays the **Open** dialog box.

This command has the following options:

- `-newwindow` — Specifies that a new window be created to display the new document. If `-newwindow` is not specified, the new document replaces the current document. `-nw` is a synonym for `-newwindow`.
- `-ok` — Opens the requested document without prompting you about unsaved changes to the current document or file. Unsaved changes will be lost. This option is ignored if `-newwindow` is used.
- `-readonly` — Opens the requested document as a read-only document.
- `-xml` — Specifies that the document to edit is an XML document. When this option is selected, a completeness check will not be performed (that is, `-nocc` is implied), and specifying `-cc` will have no effect.

- `-encoding` — Determines the encoding of the document, as specified by *string*; *string* must be one of the following strings:

  Supported Encodings

Adobe-Standard-Encoding	Shift_JIS
CNS11643	Big5
ISO-10646-UCS-2	GB2312
ISO-8859-1 to ISO-8859-11	KSC_5601
ISO-8859-13 to ISO-8859-16	EUC-JP
windows-1250 to windows-1258	CEUC
US-ASCII	TEUC
UTF-16	EUC-KR
UTF-8	cp932
cp949	cp936
cp950	

- `-untaggedascii` — Opens the requested ASCII document. SGML markup, if any, will not be interpreted. `-untagged` is a synonym for `-untaggedascii`.
- `-cc` — Specifies that Arbortext Editor perform a completeness check when it opens a Arbortext Editor document. (By default, a completeness check will not be done if the document was saved in Arbortext Editor.) Using this option is recommended when opening documents which were created in Arbortext Editor, but were later edited with another program.
- `-nocc` — Specifies that Arbortext Editor not perform a completeness check when opening a document. The document must be fully normalized according to SGML's reference concrete syntax.

  If neither `-cc` nor `-nocc` is specified, the default behavior of the `edit` command is to search for an SGML comment containing the word “PTC Inc.” in the first 2000 characters of the document instance. If the comment is found, the document is assumed to have been created by Arbortext Editor and is normalized.
- `-cd` — Specifies that Arbortext Editor should change its working directory to the directory containing the file that is opened.

  If this option is not included, there is no change to Arbortext Editor’s working directory when the file is opened.
- `-stylesheet` — Specifies the stylesheet *name* be used in place of the default stylesheet for the Edit view. *name* is the path and file name of a stylesheet file. *name* can also be the base name of the stylesheet file (the file

name without the path). If just the base name is supplied, Arbortext Editor first searches for the file in the document directory, and then the document type directory.

- `-catalog` — Specifies the name of the directory *dir* containing an interchange catalog, which provides the document entity to be edited if no path name is specified. *dir* is prepended to the current `set catalogpath` option if not already present. If the interchange catalog specifies a FOSI for the document entity, this FOSI will be used for the Edit view unless the `-fosi` option is given.
- `-last` — Displays the previously edited file.
- `-current` — Reloads the current file.

Examples:

```
edit
edit -last -ok
edit -untag mailist
edit -r report
edit -nocc userman
edit -cd C:/test.xml
```

eval (Command)

`eval` *expression1* [,*expression2* …] [output=*outspec*]

This command evaluates each expression and displays the results in the message window.

The output of each expression is separated from the next by the value of the `$OFS` variable and terminated by the value of the `$ORS` variable. By default `$OFS` is a single blank and `$ORS` is a single newline character. The expressions may be printed to a file or pipe using the `output= specifier` argument. Refer to show aliases on page 707 for a complete description of output option. Note that expressions are delimited by a comma. In this case, *outspec* can be any of the following:

- The name of a file (this can be a complete path name). A right angle bracket (>) preceding the file name causes the result of the `eval` command to be appended to the end of the file.
- An asterisk (`*`) indicating the message window. If preceded by a right angle bracket (`>*`), the output is appended to the message window instead of replacing its contents. Additional predefined message windows `msgwin2`, `msgwin3`, or `msgwin4` use the specifier `output=*2`, `output=*3`, or `output=*4` respectively.

- A question mark (`?`) preceding the output file name. If the file name starts with a question mark, the file name is a variable name whose value is set to the output produced by the command. If the second character is a right angle bracket (>), the output is appended to the current value of the variable instead of replacing it.
- An exclamation point (`!`) preceding an output specifier. This suppresses the normal prompt to confirm overwriting an existing file.

Note

Use only whole numbers within `eval` commands; the `eval` command does not perform floating point arithmetic.

Example
```
eval (3 + 4) * 5
```

execute (Command)

`execute` [*command* | { *cmd1*; [*cmd2*; …]}]

This command causes the command in the command subwindow to be performed (as does the **ENTER** key at the command line).

When used with the `command` option, it defers evaluation of variables until runtime. With the `map` on page 669, `alias` on page 617, and `print` on page 691 commands, this means that evaluation is delayed until the action is triggered.

Examples
```
exe cp $filename $filename.bak
execute {cp $filename $filename.bak; save;}
exec print composed printer="\\HP100PS\PS686" file=$print_file
```

exit

`exit`

This command closes the current window. If you only have one window open, Arbortext Editor will close saving any changes made to the document.

Examples
```
exit
```

find

`find { -panel | [-b | -f] [-select | -noselect] [-c | -noc] [-word | -noword] [-e | -noe] [-entityscan | -noentityscan] [-markup '<tagname>'|'<entity>' | -t '<tagname>' | -not] [-in 'tagname'] [-wrapscan | -nowrapscan] [-q | -noq] [-graphic | -equation | -table | -s] [-p] [-r] [-d] [/string/]} [-alias | -noalias]`

This command searches from the cursor in the specified direction (forward by default) for the first occurrence of the specified string (or the previous search string, if no string is specified). The search ignores generated text. The string you enter must be delimited by one of the standard delimiters.

To replace a string with a different string, use the `substitute` command on page 715.

- `-panel` displays the **Find/Replace** dialog box.
- `-b` searches backward.
- `-f` (the default) searches forward.
- `-select` (the default) highlights and selects the matched string.
- `-noselect` positions the cursor at the end of the search string.
- `-c` performs a case-sensitive search.
- `-noc` (the default) finds a string regardless of how it is capitalized. The `set case` on page 761 command sets this parameter for all subsequent searches.
- `-word` (or `-w`) finds a string only when it matches a whole word.
- `-noword` (or `-now`) matches strings even when they are part of a larger word
- `-e` specifies that the search string is a regular expression. This means that some punctuation characters such as `*`, `\`, and `.` perform special functions. For example, in `find -e /p..t/`, the period `.` is a placeholder representing any character. The command would find “pret”, “port”, and “part”, as well as the string “p..t”.
- `-noe` (the default) specifies that the search string is not a regular expression. The `set expressions` on page 794 command sets this parameter for all subsequent searches.
- `-entityscan` (`-es`) controls whether or not the `find` command will search the contents of entities.
- `-noentityscan` (`-noes`) will prevent the search from including the contents of entities. For example, the command `find -es /script` would search for the word “script” and include entities. Setting this option overrides the `set entityscan` on page 792 command.
- `-markup` (`-m`) indicates that the search string contains one or more tags or entities. Tags are indicated by surrounding angle brackets; an end tag contains

a slash. Entities start with an ampersand (`&`) and end in a semicolon (`;`). For example, to search for the end of an italic tag pair, you might enter `find -m '</italic>'`. To search for a copyright symbol, you might enter `find -m '&copy;'`. To search for a comment, you might enter `find -m '<_comment>'`. (Similar searches would locate processing instructions.)

If the `set balancedselections` is set to `on`, the command `find -m '<`*`tagname`*`>'` results in finding the start tag, content, and end tag of the next <*tagname*> element, with the cursor being placed directly after the end tag.

- `-t` works like the `-markup` option except it does not find entities. This option is provided for backward compatibility; scripts should use the `-markup` option.
- `-not` (the default) indicates the search string does not contain tags. The `set tagscan` on page 908 command sets the default value for this parameter for all subsequent searches.
- `-in '<`*`tagname`*`>'` searches the text within specified markup by specifying the tag name to search within .
- `-wrapscan` (or `-ws`) causes the search to wrap to the beginning of the file when the end of the file is reached. `-wrapscan` is the default.
- `-nowrapscan` (`-nows`) prevents the search from wrapping. The `set wrapscan` on page 927 command sets the default value for this parameter for all subsequent searches.
- `-q` (for “quiet”) suppresses the “string not found” message generated when a search fails. This is useful for putting the `find` command in command files or aliases.
- `-noq` (the default) displays a message when Arbortext Editor cannot find a string that matches the search string.
- `-graphic` (or `-gr`) searches forward to the next graphic in the document. Search strings are not permitted with the `-graphic` modifier.
- `-table` (or `-tab`) searches to the next table in the document. Search strings are not permitted with the `-table` modifier.
- `-s` searches for the next special element (graphic, equation, or table) in the document. Search strings are not permitted with the `-s` modifier.
- `-p` searches to the tag at the end of the current document division (the parent tag). It can be combined with `-b` to move back to the beginning of a division. Search strings are not permitted with the `-p` modifier.
- `-r` repeats the search in the opposite (reverse) direction. Search strings are not permitted with the `-r` modifier.

- `-d` searches for the mate of the square bracket, curly brace, parenthesis, or double quote directly preceding the cursor; otherwise, it searches for the closest delimiter and positions the cursor next to its mate. Search strings are not permitted with the `-d` modifier.
- `-alias` specifies that the search string can contain element aliases, but not real names, if an alias map has been applied to the document. The default is `-noalias`.
- `-noalias` indicates that the search string cannot contain element aliases, even if an alias map has been applied to the document.

The `set case` on page 761, `set expressions` on page 794, `set tagscan` on page 908, `set entityscan` on page 792, `set markupscan` on page 850 and `set wrapscan` on page 927commands can modify the behavior of the `find` command.

If you use the `find` command in a script, be sure to use a valid string delimiter for the search string. You must supply either a matching close delimiter or place the string at the end of a line. For instance, `find /therefore` if it is the end of a line and `find /therefore/` are both valid. Using `{find /therefore}` as part of another command specification isn't valid because the curly brace is a command group delimiter, not a string delimiter. By supplying the closing delimiter `/`, the command specification `{find /therefore/}` is now valid.

Examples

```
find -b /and
f -f /and
f /therefore
f -c /Therefore
f -noc /therefore
f -e /gr.y
f -noe /grey/
f -m '</emphasis> hunting'
f -m 'Our Company Name&copy;'
f -not /hunting/
f -p
f -r
f -d
```

When the search text does not contain tags, the `-markup` option is not specified, or the `set markupscan` preference is `off`, then inline tags and processing instructions are ignored when matching text. If a tag starts or ends a line, it is considered a word boundary and also limits a text match. For example, the search text `two words` would match `two <emphasis>words</emphasis>`, but would not cross paragraph boundaries to match the following:

```
two</para>
<para>words
```

Elements suppressed by the tag_display command on page 719 can be recognized and operated on by the find command by setting set hiddentagscan on page 829 to on.

find_attr_string

find_attr_string [*string*]

This alias searches forward from the cursor in the current document for the next occurrence of an attribute value matching *string*. If you do not supply the argument, a response panel prompts you to do so.

For instance, if you wanted to find a paragraph tag with an id of **p100**, you can omit the p and use 100 as the string and the find_attr_string command would still find it. If the attribute exists, the document text moves to show the highlighted tag and the cursor is placed to the right of the tag to facilitate modification.

fas is a synonym for find_attr_string.

Examples

```
find_attr_string 100
fas drw
fas 'diag.tif'
```

find_attr_value

find_attr_value [*value*]

This alias scans forward from the cursor in the current document for the next attribute whose value exactly matches *value*. If you do not supply the argument, a response panel prompts you to do so.

For example, if you have <xrefs idrefs="abc def">, the command find_attr_value “abc def” would find it. The command find_attr_value “abcdef” would not find it because the attribute values don't match exactly.

Another command you could use in this situation is find_id. find_id abc would find the IDREFS because find_id recognizes attribute values delimited by spaces.

fav is a synonym for find_attr_value.

Examples

```
find_attr_value p100
fav 1
fav "1pc+2pt"
```

find_entity

`find_entity` *entity*

This alias searches forward from the cursor in the current document for the first occurrence of the specified entity reference.

`fe` is a synonym for `find_entity`.

Examples

```
find_entity bullet
fe ndash
fe alpha
```

find_id

`find_id` [*ID* | *IDREF*]

This alias scans forward from the cursor in the current document for the next element whose ID or IDREF attribute matches *ID* or *IDREF*.

The search can recognize different parts of an ID attribute if they are separated by delimiting spaces. If you do not supply the argument, a response panel prompts you to do so.

For example, if you have `<xrefs idrefs="abc def"`, the command `find_id abc` would find it. (Note that the `find_attr_value abc` command would not find it because the attribute values don't match exactly.)

`fid` is a synonym for `find_id`.

Examples

```
find_id abc
fid p100
fid center
```

find_pi

`find_pi`

This alias searches forward from the cursor in the current document for the next processing instruction, highlights the processing instruction tag, and moves the cursor to the right of the tag to facilitate modification.

`fp` is a synonym for `find_pi`.

Examples

```
find_pi
fp
```

find_pi_string

`find_pi_string` [*string*]

This alias scans forward from the cursor in the current document (and wraps around) for the next occurrence of a processing instruction tag having a value matching *string*. If you do not supply the argument, a response panel prompts you to do so. If the string exists, the processing instruction tag is highlighted and the cursor is placed to the right of the tag to facilitate modification.

`find_pi_string` is the quickest way to find specific processing instruction tags or names of style options, where *string* is the tag name or style option.

Suppose you want to find a `_kern` processing instruction tag with attribute value set to `3pt`. Use the following method:

1. Insert an example of a processing instruction tag with the same attribute values as the tag you want to find (for example, `_kern` tag modified to `3pt`). Highlight this tag.
2. At the command line, type: `eval $selection`.
3. Press **ENTER**. An evaluation panel displays the value of the highlighted tag:
   `<!Pub _kern Amount="3pt"`
4. At the command line, type: `fps` followed by a space and the result from the `eval $selection` output (minus the delimiting angle brackets and initial question mark) surrounded by single quotes. For example:
   `fps 'Pub _kern Amount="3pt"'`
5. Delete the example tag you inserted before pressing **ENTER**.

`fps` is a synonym for `find_pi_string`.

Examples

```
find_pi_string newpage
fps italic
fps 'hyphen-point'
fps 'Pub _kern'
fps 'Pub _texmac Tex="$\circ$"'
```

find_pi_value

`find_pi_value` [*value*]

This alias searches forward from the cursor in the current document for the next processing instruction with a value exactly matching *value*. If you do not supply the argument, a response panel prompts you to do so. If the string exists, the processing instruction tag is highlighted and the cursor is placed to the right of the tag to facilitate modification.

To determine the value for value, you can use the following method. For example, to find a string of `3pt` within a `_kern` processing instruction tag:

- Insert an example of a processing instruction tag with the same attribute values as the tag you want to find (for example, `_kern` tag modified to `3pt`). Highlight this tag.
- At the command line, enter:
  `eval $selection`
- Press **ENTER**. An evaluation panel displays the value of the highlighted tag.
- At the command line, type: `fps` followed by a space and the result from the `eval $selection` output (minus the delimiting angle brackets and initial question mark) surrounded by single quotes. For example:
  `fpv 'Pub _kern Amount="3pt"'`
- Delete the example tag you inserted before pressing **ENTER**.

Examples

```
find_pi_value 'Pub _newpage'
fpv 'Pub _font Posture="italic"'
fpv 'Pub _hyphen-point'
fpv 'Pub _kern Amount="2pt"'
fpv 'Pub _texmac Tex="$\circ$"'
```

for

`for` (*expression1*; *expression2*; *expression3*) {*command-list*}

`for` ($*variable* in $*array*) {*command-list*}

The first form of this command evaluates *expression1* and then executes the listed commands while *expression2* is true. At the end of each iteration, executes *expression3*. This is equivalent to:

expression1 `;` `while (`*expression2*`) { `*command-list* `;` *expression3* `;}`

The second form of this command executes listed commands with the variable set to each subscript of the array. The subscripts are returned in order (from the lowest to the highest) for a normal array. If the array is associative, the subscripts are returned in arbitrary order. The command list must not contain another `for` ($variable in $array) statement over the same array.

Examples

```
# Display each of the variables in the
#array "$includes" on std out.
for ($i = 1; $i <= count($includes); $i++) {
eval "[$i], $includes[$i]" output=-;
}
# This alias creates a list of all valid tags
# in the current document:
tag_names($tags);
$list = " "
```

```
for ($i in $tags) {
 $list .= delete($tags[$i]) . " "
}
```

format

`format` [continue | onepass | allpasses | layout | quit | stop | help | tokendump | insert *string* | delete *n*] [force | auto] [wait | nowait]

Use the format command to control the formatting process. This command is not supported for the compact installation of Arbortext Editor.

The following primary options are available.

- `format continue` (the default) resumes or starts formatting. If formatting stopped due to an error, Print Preview will resume with the page containing the error.
- `format onepass` makes one pass to reformat the entire document.
- `format allpasses` formats the document as many times as is necessary to complete auxiliary entities such as citations, indices, and tables of contents.
- `format layout` formats the document in the same way as the `allpasses` command and creates a `layout` file in addition to the `dvi` file.
- `format quit` ends document formatting and unloads the formatter. The formatter will be reinitialized and reloaded if formatting is requested again.
- `format stop` suspends document formatting; formatting can be resumed later with `format continue`.

The following commands are valid only when a preview or print job has been stopped by an error. They're useful for debugging formatter errors (as directed by Technical Product Support):

- `format help` which requests a help message from the formatter.
- `format tokendump` which displays what the formatter was processing when the error occurred.
- `format insert` *`string`* which sends the specified string to the formatter for interpretation.
- `format delete` *`n`* which deletes *n* tokens.

The `auto` and `force` options are mutually exclusive and control whether Arbortext Editor decides if formatting is necessary.

- The (default) *auto* option lets Arbortext Editor decide whether the document needs to be formatted or not. Arbortext Editor will trigger formatting if the document has not been formatted yet during the current session, or if the previous formatting command left the document in the `refmt-needed`

state, or if the document has been modified since the preceding formatting command. If the document instance is up to date with regard to formatting, then `format auto` will not reformat the document instance.

- The *force* option means to perform at least one formatting pass regardless of whether Arbortext Editor thinks it is necessary.

The *wait* and *nowait* options control whether Arbortext Editor should wait for the format to complete before continuing to execute commands.

Note

The *nowait* option is the default. However, if you are running Arbortext Editor with a `-c` option from the command line, Arbortext Editor will always wait for the print to complete, even if you specify the *nowait* option.

- The *wait* option instructs Arbortext Editor to wait until the format process is complete before continuing with subsequent commands. If formatting fails, your function can return a non-zero error status and retrieve the error message from the variable *main::ERROR*.
- The *nowait* option instructs Arbortext Editor to continue processing and perform the formatting in the background. This is what usually happens when you choose **File ► Print Preview** or **File ► Print Published**.

`fmt` is a synonym for the `format` command.

Examples

```
format
fmt cont
fmt onepass
fmt allpasses auto
fmt allpasses force wait
fmt quit
fmt stop
fmt help
fmt tok
fmt ins '\showthe\hsize'
fmt delete 5
```

global

`global` *var1* [, *var2* …]

The `global` statement (command) declares one or more scalar or array global variables in the current package. It has the same syntax as the `local` statement. A scalar variable can be given an initial value by assignment using the form var = expr; see the examples that follow. If no initial value is given, the variable is initialized to null.

An array is declared by appending "[]" to the variable name. In this case the array can be a normal or an associative array, depending on context and will grow as necessary. A fixed size normal (that is, non-associative) array may be declared by specifying the size of the array inside the brackets, for example, *arr*[10]. The size may be an expression, which is evaluated at run time. The size is also the upper bound of the array since the first subscript is 1. To declare a fixed size array with a different minimum subscript, use the form

```
global arr[lb..hb]
```

where `lb` and `hb` are expressions giving the low and high bounds of the array. Arbortext Editor will signal a run-time error if you attempt to access elements outside the bounds of a fixed-size array.

Note

`global` is an executable statement. Each time it is encountered it will reinitialize the global variable.

Examples

```
global doc_name
global win = -1, doclist[]
global n1 = -5, n2 = n1 + 10
global arr[n1..n2]
```

help

`help` [`-2` | `-3` | `-4` | `-msg` | `-msgwin2` | `-msgwin3` | `-msgwin4`] [`-title` '*windowtitle*'] [`-geometry` *WxH+X+Y*] [`-tag`] [`-file`] [`-mouse` | *topic*]

Typing `help` at the command line, without arguments, opens the Arbortext Editor Help Center and displays its default topic.

Supplying the various options opens the Help Center to a specific topic, opens tag help, or opens other ACL application windows as follows. (Only the `-topic` option affects Help Center.)

- The numeric arguments `-2`, `-3`, `-4`, `-msg`, `-msgwin2`, `-msgwin3`, and `-msgwin4` refer to choices for multiple tag help or ACL application windows. These windows can be used to view multiple help messages at one

time. They are also available to application programmers writing their own help.

- `-title` lets you override the default window title with a name you supply within quotation marks. If *windowtitle* is not surrounded by quotation marks, the remainder of the command line is used as the title.
- `-geometry` specifies the size and position of the desired help window. It is a string of the form *WxH+X+Y*, where *W* is the width, and *H* is the height of the window in pixels. *X* and *Y* specify the position of the window relative to:
  - the upper left (corner of the screen) if *+X+Y* is used;
  - the upper right if *-X+Y* is used;
  - the lower left if *+X-Y* is used;
  - and the lower right if *-X-Y* is used.

  If the help window is currently displayed, its size and/or position can be changed according to the geometry specification.
- `-tag` specifies that *topic* is the name of an element (tag) in the document type associated with the current document. The `-tag` option should be given to get help for a tag with the same name as the Arbortext Editor command (for example, `index`).
- `-file` allows you to specify an alternate Arbortext Editor help file or custom `.chm` help file. For example, if you want to open the Arbortext Change Page for Defense help file, you would specify:
  ```
  help -file cpa.chm
  ```
- `-mouse` displays a help message based on the context of the mouse arrow.
- *topic* can be command names (for example, `help set`), alias names (for example, `help find_id`), ACL topics (for example, `help functions`) or other terms. If *topic* is not recognized as a command, alias, or specific topic, the Arbortext Editor Help Center will open with the search results for *topic*.

`hel` is a synonym for `help`.

Examples:

```
help
help find
help declare_graphic_entity
hel -2
hel -msg -title 'Set Commands' set
hel expressions
help operators
hel logical expressions
help operands
help -tag link
```

if

`if` (*condition*) {*cmds*} [else if (*condition*) {*cmds*} …] [else { *cmds*}]

This command executes the commands listed only if the given condition is true. `else if` clauses let you specify alternative commands to be executed when alternate conditions are true. The `else` clause specifies a course of action to be taken if none of the conditions tested for are true. You may specify any number of `else if` clauses, but only one `else` clause.

cmds is a list of one or more commands. The list must be enclosed in curly braces. *condition* is a logical expression describing the case in which these commands are to be executed.

Examples

```
if ($doctype=="book") {
 dft glossword font-change;
 ms -global glossword TypeStyle=bold
}
if ( $answer == "yes" ) {
 # do something
}
else if ($answer == "no") {
 # do something else
}
else {
 # try again
}
if (! access($file,"w") {
 message "can't write file $file"
}
else {
 write -buffer buf1 $file
}
copy_mark
if ($status !=0) {
 message "You must select something first"
}
```

insert_accent

`insert_accent` *accentname*

This command places the specified accent on the first item to the left of the cursor if the item is a character. *accentname* is an entity name. Not all accent names are valid in all document types. The DTD for the document must define the appropriate accent entity as given in the following table.

Accents

Accent name	Entity name	Accent name	Entity name
acute*	acute	dotbelow	dotbelow
barbelow	barbelow	grave*	grave
breve	breve	double acute	dblac
caron	caron	macron*	macr
cedilla*	cedil	ring*	ring
circumflex*	circ	tilde*	tilde
dotabove	dot	umlaut*	uml
*Accents marked with an asterisk are ISO-Latin-1 characters and will display as accented letters in the Edit pane. The other accents are available in PostScript printer fonts and will print but are shown as SGML entity references in the Edit pane.			

The entity name may also be used as the *accentname* argument.

is a synonym for `insert_accent`.

Examples

```
insert_accent acute
ia macron
```

insert_column

`insert_column` [before]

This command inserts one column after (or before if before is given) the column containing the cursor in the current table cell. This command is not valid if the cursor is not currently in a table cell.

Examples

```
insert_column
insert_column before
```

insert_entity

`insert_entity` [*name*]

This alias inserts the specified entity reference at the current cursor location, prompting for an entity name if none is supplied. The specified entity name must not be enclosed in quotation marks.

In the Edit view, the cursor is placed after the inserted entity. In the Document Map view, the cursor is placed before the inserted entity; you can control this behavior with the `set docmappastecaret=off` option.

is a synonym for `insert_entity`.

Examples
```
insert_entity
ie ati
```

insert_equation

`insert_equation` [display | inline]

This command begins a new equation of the desired type at the cursor. The default is `display`.

Examples
```
insert_equation
insert_equ inline
```

insert_graphic

`insert_graphic` [*tagname*]

This command inserts the graphic tag specified by *tagname* (for this tag) at the cursor. If *tagname* is not specified, the first, or primary, graphic tag listed in the DCF file is assumed.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

If you are using the entity method of graphic support, you may wish to use the `insert_graphic_entity` on page 660 command alias instead.

Note

The `insert_graphic` command also uses the `insert_tag` on page 997 callback. If you add your own `insert_tag` callback after the standard callback performed by the `insert_graphic`, see the `insert_tag` callback information in the `doc_add_callback` on page 980 function help topic.

is the abbreviation for `insert_graphic`.

Examples
```
insert_graphic
```

```
ig symbol
ig piechart
```

insert_graphic_entity

`insert_graphic_entity` [*name* [*tagname*]]

This alias:

- Inserts the tag specified by *tagname* for an element identified as a graphic element in the DCF file.
- Inserts the specified entity *name* in the first value field, prompting for an entity name if none is supplied.

Note

If you are using the path name method of graphic support, you may wish to use the insert_graphic on page 659 command instead.

is an abbreviation for `insert_graphic_entity`.

Examples

```
insert_graphic_entity
ige turbogen
ige caution symbol
```

insert_include

`insert_include` [*path*] [*parse*]

This alias inserts an XML inclusion to the specified path at the current cursor location, prompting for a path if none is supplied.

The optional *parse* can be '`xml`' (the default if not supplied) or '`text`', and populates the *parse* attribute on the *include* tag itself.

insert_marked_section

`insert_marked_section` [*status1* [*status2* …]]

This command inserts a marked section around the selected region, if any, with the specified status keywords. Status keywords may be references to declared marked section parameters or may be marked section keywords.

is a synonym for `insert_marked_section`.

Examples

```
insert_marked_section %german
ims %english TEMP
ims %computer INCLUDE CDATA
ims INCLUDE
```

insert_pi

`insert_pi` [*processinginstruction*]

This alias inserts a processing instruction for non-Arbortext Editor applications, prompting for the processing instruction if none is supplied. In Arbortext Editor, a processing instruction is represented with a `_pi` markup icon that is locally modified to carry the instruction. Single quotes around a parameter preserve two or more embedded spaces.

Examples

```
insert_pi
insert_pi backslant
insert_pi 'a b'
```

The following command will hide generic processing instructions regardless of the tag display mode: `tag_display -none _pi`

To insert a Arbortext Editor processing instruction, use the `insert_tag` command.

insert_row

`insert_row` [before]

This command inserts one row after (or before if `before` is given) the row containing the cursor in the current table cell. This command is not valid if the cursor is not in a table cell.

Examples

```
insert_row
insert_row [before]
```

insert_string (Command)

`insert_string` [`-pendingdelete` | `-nopendingdelete`] [`-markup`] [`-buffer` *buffername* | `-paste`] [`-append`] /*newtext*

This command inserts *newtext* at the current cursor location.

Note

Even if you have applied an alias map to the document, you cannot specify aliases in the *newtext* string.

If there is selected text in the document, the command replaces the text according to the current setting of the `set pendingdelete` on page 866 command. You can override or force the `insert_string` command to replace or preserve the current selection with the `-pendingdelete` and `-nopendingdelete` options. If `-pendingdelete` is set, the string replaces any selected text. If `-nopendingdelete` is set, any selected text is preserved and the inserted text is added to the right of the selected text.

If `-markup` is specified, the new text is interpreted as an XML (or SGML) string. This allows you to insert markup, diacritics, or special structures such as tables.

The `insert_string -markup` command will interpret the processing instruction `<?Pub Caret>` to set the cursor to a specific position within the inserted fragment. For example, after the command: `insert_string -markup "<par>abc</par>"` is executed, the cursor is placed after the end `</par>` tag.

However, with the `insert_string -markup "<par>ab<?Pub Caret>c</par>"` command, the cursor is positioned after the letter `b`. The `<?Pub Caret>` processing instruction must come after a character or tag in the markup string. It is ignored if it occurs at the start of the string. To position the cursor before an opening tag, the form `<?Pub Caret1>` can be used, for example, `insert_string -markup "<par><?Pub Caret1>abc</par>"` would position the cursor before the starting `<par>` tag.

If the `<?Pub Caret>` processing instruction is not included in the string, the cursor is placed per the following rules:

- If the cursor is in the Edit view, the cursor is placed after the inserted string.
- If the cursor is in the Document Map view, the cursor is placed before the inserted string; you can control this behavior with the `set docmappastecaret=off` option.

The `-buffer` option places *newtext* in a named paste buffer; `-paste` puts it in the default paste buffer. The `-append` option causes the string to be appended to the current contents of the buffer; otherwise, the string replaces whatever is in the buffer.

is a synonym for `insert_string`.

Examples:

```
insert_string /Note: /
```

```
is -pd /therefore/
is -markup /e&acute;lan/
is -markup "&ndash;"
is -markup "&mdash;"
is -buffer trademarkinfo /Arbortext Architect \
 is a trademark of PTC/
```

insert_table

`insert_table`

The `insert_table` command opens the **Select a Table Model** dialog box in which you can choose a table model if the following conditions are true:

- If your document type supports multiple table models.
- `set prompttablemodels` is set to `on`.
- The cursor is in a location in which it is valid to insert more than one table model.

Once you have chosen a table model or if your document type only supports one table model, the `insert_table` command opens the **Insert Table** dialog box (or the **Insert HTML Table** dialog box if you selected **HTML** as the table model). Specify the table configuration options in this dialog box.

Example

```
insert_table
insert_tab
```

insert_tag (Command)

`insert_tag` { [`-pendingdelete` | `-nopendingdelete`] *tagname* | [`-type`] [`-panel`]}

This command inserts the specified tag or tag pair (*tagname*) around the cursor or around currently selected text, if any. Note that paired tags are inserted as a pair.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

If typed alone at the command line, `insert_tag` (or simply) brings up the **Insert Markup** dialog box that provides a list of tags that are valid at the cursor. The panel is context-sensitive meaning the tags in the panel will change as you move the cursor.

If you have applied an alias map to your document and **Tags** is the selected **Mode**, the list in the **Insert Markup** dialog box will display aliases, rather than real names, for tags that have been assigned an alias.

You can keep the **Insert Markup** dialog box on-screen and use it across editing windows.

Note

After performing an `insert_tag`, cursor placement depends on whether the inserted tag has required elements, and on the setting of the `set addrequiredtags` on page 745 option.

Note that paired tags are inserted around a highlighted selection regardless of whether pending delete is on or off. However, if pending delete is on, the default for single (unpaired) tags is to replace highlighted text. If no arguments are given in the `it` command, then `-panel` is assumed and simply brings up the **Insert Markup** dialog box.

For `insert_tag -[type]-[panel]`, the command specifies the initial markup type to be displayed in the **Insert** dialog box. Replace `-type` with one of the following: `-dtd` (for tags defined in the DTD), `-usertags` (for user-defined tags), `-pi` (for processing instructions), `-text` (for text (general) entities), `-file` (for file (external) entities), or `-ms` (for marked section parameter entities).

is a synonym for `insert_tag`.

Examples

```
insert_tag
it comment
it -pd tabstop
it -nopd enspace
```

invoke_processor

`invoke_processor`

This command invokes the table processor. Place the cursor to the right of a table tag to bring up that processor with the `invoke_processor` command.

Examples

```
invoke_processor
inv
```

join

`join`

This command joins two adjoining elements of the same type. For this command to work, the cursor must be positioned between the end tag of the first element and the start tag of the second element, or just inside the start tag of the second element. This command also works when more than two adjacent tags of the same type are selected in the Edit view.

js

`js` *JavaScript-expression*

This command passes *JavaScript-expression* to the JavaScript interpreter. The JavaScript expression is passed as a single string argument to the `javascript` function on page 396 or the `jscript` function on page 397 (depending on the `set javascriptinterpreter` on page 842 setting) for evaluation. Unlike other ACL commands, no variable substitution is performed.

Note

A semicolon is considered part of the JavaScript expression and does not terminate the command.

If JavaScript returns a non-null defined result, it is displayed in the status line in the Edit pane. The JavaScript expression is evaluated in the global shared scope.

For example,

```
js Application.activeDocument.documentElement.tagName
js print("Hello" + " world!")
js Application.alert("Hello world!")
```

link

`link` [caret | mouse] [f | b]

`link` [`-uri` *uri* | `-idref` *idref*] [`-data` *data*]

This command traverses a link in the Edit view and executes any commands associated with the link. There are two forms of the `link` command. The first form is associated with `link` and `linkuri` callbacks in the `doc_add_callback` function. The second form is associated with `linkto` callback in the `doc_add_callback` function.

For the first form, the tag type determines the action taken. Examples are shown in the following table:

Link Command Actions

Tag type	Action
`_link` processing instruction	Attributes on the `_link` processing instruction determine the nature of the link.
Element with a non-blank `ID`, `IDREF`, or `IDREFS` attribute	Moves the caret forward to the next use of that ID, wrapping if necessary. If a tag has more than one ID name in its attribute values (for example, `IDREF` field with multiple values, or an `IDREF` field plus an `ID` field and both have values), then you are prompted with a list of IDs from which to select. If you link from a tag with one ID, to a tag with multiple IDs, then another tag with multiple IDs, you can be linked to the next use of the ID originally linked to. This lets you cycle through all the references to a given ID without getting hung up on tags that also reference other IDs.
Element declared in the `.dcf` file with an `IDREF` or `URI` attribute	For `IDREF` attributes, moves the caret to the target. For `URI` attributes, opens the user's Web browser and displays the URI.
file entity	Opens a separate Edit pane in which you can edit the file entity.
graphic	Opens an associated application, in which you can edit the graphic.
equation	Opens an associated application, in which you can edit the equation.

Following is a description of `link` command parameters for the first form of the command:

- `f` (the default) — Traverses forward, establishing a new link to a target.

  In this case, the `link` command gets link data from the tag to the left of the cursor if the `caret` option is used, or from the tag underneath the mouse pointer if the `mouse` option is used. `caret` is the default.
- `b` — Traverses backward to the previous location.

  The cursor is moved back to the tag from which the most recent link occurred. The location of the mouse or cursor does not affect the `link` command when linking backwards.

Following is a description of `link` command parameters for the second form of the command:

- `-uri` — Explicitly provides a URI value for the target for the link.
- `-idref` — Explicitly provides an IDREF value for the target for the link.
- `-data` — A string value that is passed to the `linkto` callback.

  The data provided for the link can be anything relevant to the operation performed by the callback.

Examples

```
link
link mouse
link b
link -idref help1723
```

load_buffers

`load_buffers` [*directory*]

This command loads named paste buffers from the specified directory. If no directory is given, the current document directory is used. Buffers are saved with the `save_buffers` on page 706command.

is a synonym for `load_buffers`.

Examples

```
load_buffers
lb /ati/doc/publ/text
```

local

`local` *var1* [, *var2* …]

The `local` statement (command) declares one or more scalar or array variables local to the current block, which ends with the enclosing '}'. It has the same syntax as the `global` statement.

A scalar variable can be given an initial value by assignment using the form `var = expr`; see the examples that follow. If no initial value is given the variable will be initialized to null.

An array is declared by appending "[]" to the variable name. In this case the array can be either a normal or associative array depending on context and will grow as necessary. A fixed size normal (that is, non-associative) array may be declared by specifying the size of the array inside the brackets, for example, arry[10]. The size may be an expression, which is evaluated at run time. The size is also the upper bound of the array since the first subscript is 1. To declare a fixed size array with a different minimum subscript, use the form

`local` *arr*[*lb*.. *hb*]

where *lb* and *hb* are expressions giving the low and high bounds of the array. Arbortext Editor will report a run-time error if you attempt to index outside the bounds of a fixed-size array.

Note

`local` is an executable statement, so that initial values get evaluated each time the command is executed. This also means that for best efficiency it is better to not use `local` commands inside loops which execute many times.

It is an error to declare a local variable which was previously defined at the current level. See the following sample function.

The `local` command is only allowed inside function definitions.

Examples

```
function f(x, src[])
{
 local doc, save_doc = current_doc()
 local assoc[]; # declare associative array
 local dest[count(src)];# declare dest same size as src
 local x; # illegal, hides parameter
 local m=1
 {
 local m=2; # ok
 local x[]; # ok
}
 local m; # illegal, hides previous value
}
```

lookup

`lookup` [*word*] [output=*filename*]

This command displays definition or alternate spellings for the word supplied. Words that contain spaces must be enclosed in quotation marks. If no word is supplied, the definition or spellings displayed are for the word currently selected (or the word nearest the cursor, if there is no selection). Selections can be made from windows other than Arbortext Editor windows, as well. Words are checked against the Proximity/Merriam-Webster Linguibase. Note that the language used for the dictionary is determined by the position of the cursor in the document. You can set a language on individual tags in a document, so that language can vary based on the position of the cursor.

If *output* is specified, `lookup` writes the definition of the word to *filename*. Note that *filename* can be any of the following:

- The name of a file (this could be a complete path name).

  A right angle bracket (>) preceding the file name causes the definition to be appended to the end of the file. An exclamation mark (`!`) preceding the file name causes the file, if it exists, to be overwritten without prompting for confirmation.
- An asterisk (`*`) indicating the message window.

  If preceded by a right angle bracket (`>*`), then the output is appended to the message window instead of replacing its contents. Additional predefined message windows `msgwin2`, `msgwin3`, or `msgwin4` use the specifier `output=*2`, `output=*3` or `output=*4`, respectively.
- A dash (`-`) indicating standard output. (This is normally the window from which you start Arbortext Editor.)
- A question mark (`?`) preceding the output file name.

  If the file name starts with a question mark, then the file name is actually a variable name whose value is set to the output produced by the command. If the second character is a right angle bracket (>), then the output is appended to the current value of the variable instead of replacing it.

Examples

```
lookup
loo set
loo bicycle
loo "fork over"
```

map

`map` [*window* | all] *keyname* { *cmd1* | { *cmd1*; [*cmd2*; …]}}

This command assigns command functions to keyboard characters and mouse buttons for the window specified. The value for *window*, which is a window class name or user-defined keyboard shortcut, can be any of the following:

- `edit`, which designates the Edit pane.
- `cmd`, which designates the command line.
- `helpwin` (also called `helpwin1`), `helpwin2`, `helpwin3`, `helpwin4`, `msg` (also called `msgwin1`), `msgwin2`, `msgwin3`, and `msgwin4` designate multiple Help windows. Use `msgwin` for Arbortext Editor messages and output from the `show`, `eval`, and similar commands.
- *window* name as returned by the `window_name` function.
- `@`*name*, which designates the name of a user-defined keyboard shortcut.

If *window* is a class name, the mapping affects all windows using the specified class keymap, otherwise it changes the mapping in the keymap defined by a previous `define_keymap` or `copy_keymap` command.

A list of several window classes can be made by using a plus sign (+) to separate the names. To subtract a window class from a list, use a hyphen (-). The option `all` is a synonym for `edit+cmd`. Note that the `helpwins` and `msgwins` are not included in the `all` option. To map a key for all possible window classes, use: `map all+textwins`. The `textwins` option is a synonym for `helpwins+msgwins`. The `helpwins` option is a synonym for `helpwin1+helpwin2+helpwin3+helpwin4`. The `msgwins` option is the same as `msgwin1+msgwin2+msgwin3+msgwin4`.

The default keymap is the one attached to the current window (normally the `edit` class user keymap).

The value for *keyname* can be any one of the following:

- any of the letters `a-z` or `A-Z`,
- any of the punctuation characters on the keyboard,
- any of the digits `0-9`,
- any of the function keys **F1**, **F2**, ... **F35**

  Synonyms, or alternate key names, are in parentheses. `Keypad` can be replaced by `KP` or `Num` (for example, `KP_9` is equivalent to `Keypad_9`).
- any of the keypad keys:

  Punctuation Characters On the Numeric Keypad

Symbol	Name	Synonym
(	Keypad_ParenLeft	Num_(
)	Keypad_ParenRight	Num_)
*	Keypad_Multiply	Num_*
+	Keypad_Add	Num_+
,	Keypad_Separator	Num_,
-	Keypad_Subtract	Num_-
.	Keypad_Decimal	Num_.
/	Keypad_Divide	Num_/
=	Keypad_Equal	Num_=

- any of the following name keys.

  Abort (Stop)
  Again (Redo)
  Alternate
  LineFeed
  Mark (Select)
  Move

BackSpace
BeginLine
Break
Cancel (Stop)
CharDel (DeleteChar)
Clear (ClearDisplay)
ClearDisplay
ClearLine
CmdKey
Compose
Copy
Cut
Del (Delete)
Delete
DeleteChar
DeleteLine
Do (Menu)
DownArrow
DownBox
Edit Key
End (EndLine)
EndLine
Enter (Return)
Esc (Escape)
Escape
Execute
Exit
F1, F2, ... F10
F11, F12, ... F20
F21, F22, ... F35
Find
Grow
Hold (Pause)
Help
Ins (Insert)
Insert
InsertHere (Insert)
InsertLine
LeftArrow
LeftBox
LineDel (DeleteLine)
LineDelete (DeleteLine)
MultiKey
Next
NextScreen (Next)
NextWindow (Next)
NextWndo (Next)
NumLock
PageDown (Next)
PageUp (Previous)
Paste
Pause
PF1
PF2
PF3
PF4
PgDn (Next)
PgUp (Previous)
Pop
Prev
PrevScreen
Previous
Print
Prior (Previous)
R1, R2, ... R16
Read
Redo
Remove
Repeat
Reset
RightArrow
RightBox
Save
Select
Shell
Space
Stop
System
Tab
Undo
UpArrow
UpBox
User

Caution

Remapping the **ENTER** key at the command line can make it impossible to execute further commands. However, you cannot remap **CONTROL**+**SHIFT**+**ALT**+**ENTER** , which can always be used to execute commands at the command line.

Commands can also be mapped to the mouse buttons. `M1` and `M2` designate the left and right mouse buttons on a two-button mouse. On a three-button mouse, `M2` is the middle button and `M3` is the right button. However, to use a three-button mouse on Windows, you also need to set the environment variable *APTMOUSE* to equal `3`. Any of the shift or control sequences may be used with any mouse button. In addition, the designation for the mouse button may be prefixed with `Up-`, `Double-`, or `Triple-` where:

- `Up` maps a command to the release of the designated mouse button (commands not prefixed by "Up" are executed when the button is first pressed down)
- `Double` maps a command to the second click in succession of the designated mouse button
- `Triple` maps a command to the third click in succession of the designated mouse button

The `map` command accepts the abbreviations `ctrl` for `Control`, `dbl` for `Double`, and `trpl` for `Triple`. Note that mouse commands are additive; thus, before a command mapped to the third click of the right mouse button (Triple-M3) is executed, any commands mapped as M3, Up-M3, Double-M3 and Up-M3 would be executed, followed by Triple-M3 and finally Up-M3.

Commands can further be mapped to scroll wheel actions with `scrollwheel`. You must preface scroll wheel actions with either `down` or `up` to indicate whether the action is for scrolling the wheel down or up. For example, `down+scrollwheel` indicates the mapped command is triggered by scrolling the wheel down.

Examples

```
map BackSpace caret 0,-1
map all f1 caret next
map shift-f2 print unformatted screen \
 header=none
map control-shift-alt-q quit ok
map control-d delete_character
map shift-alt-d {caret 0,+1; delete_character;}
map shift-M3 {caret mouse; menu;}
map shift-M1 caret mouse
map @mymap Control-y paste
map Double-M3 help
```

```
map all+textwins-cmd shift-UpArrow window -1,0
map all+textwins down+mousewheel ScrollWheelDown
map all+textwins down+mousewheel ScrollWheelDown
map all+textwins up+mousewheel ScrollWheelUp
map edit+textwins control+down+mousewheel ZoomDown
map edit+textwins control+up+mousewheel ZoomUp
```

mark

`mark` [`-selection` | `-noselection`] [`-invert` | `-noinvert`] [`-word`] [begin | end]

This command starts or terminates a highlighted selection at respective cursor positions. In cases where neither `begin` nor `end` is specified, `begin` is assumed unless a selection is already started, in which case `end` is assumed.

In general, `-selection` (the default) causes Arbortext Editor to claim ownership of the window system PRIMARY selection. The window system selection is not affected if `-noselection` is specified. However, if `end` is also given with `-selection`, then the primary selection is affected only if a non-empty region is marked; if the marked region is empty, then `mark -select end` clears the marking and the window system PRIMARY selection is unaffected.

The `-invert` and `-noinvert` options determine whether selected text can be highlighted on the screen.

The `-word` option causes the selection to be extended in whole word increments.

Examples

```
mark
mar begin
mar end
# This is the default mapping for the left mouse button
# It enables you to position the caret in
# another window without losing the selection.
map all m1 {clear_mark;caret mouse;mark -noselect begin;}
map all up-m1 {mark -select end;}
mar -noinvert
```

menu

`menu`

This command opens a shortcut menu. The location of the cursor within your document determines which shortcut menu opens. For example, if the cursor is within a table, the `menu` command opens the Table shortcut menu.

Examples

```
menu
men
```

menu_add

`menu_add` [`-before`] *destination label* [`-separator` | [`-cmd` {*cmds*}] [`-key` *keyname* | `-keytext` "*text*"] [`-toggle` '*togexpr*'] [`-radio`] [`-active` "*actexpr*"]]

This command adds an item to an existing menu where *label* is the name for the new menu item and *destination* is a menu path indicating the existing item that the new item should follow.

If *destination* or *label* contains spaces or special characters, it must be enclosed in quotes. Please note that underscores, spaces, and periods are not allowed for menu bar labels.

If `-before` is specified, the new item will appear before the destination item rather than after it.

Note

This command adds items to existing menus. To create a new menu, use the `menu_add -menu` on page 676 command.

label specifies the label displayed for the item. If *label* contains any variable references, for example, `"Modify $tagname"`, the variable's value is substituted into the label string each time the menu containing the item is posted. Note that the label string *label* must be enclosed in single quotes to prevent the command parser from expanding the variable reference when the command is parsed.

A mnemonic access key may be specified for the menu item by preceding the desired letter in *label* with `&`. To use a literal `&` in *label*, use two ampersands, for example, `"Save && Exit"`.

Tip

Ensure that you use the correct values for menu items, by referring to the file *`Arbortext-path`*`\lib\editmenu.cf`.

If a value uses a variable, enclose it in single quotes (rather than a double quote) inthis command.

If the menu item is not declared correctly, your `menu_add` command may fail with an error message of the type shown below:

```
[A11204] No menu matched: .Edit.&Undo Delete
```

`-cmd{ cmds }` indicates the commands to be executed when the new menu item is selected. If an item is mapped to more than one command, the commands must be enclosed in curly braces and separated by semicolons.

`-key` *keyname* specifies the accelerator key (used for keyboard shortcuts) displayed in the right-hand field of the menu item. To take effect, the accelerator key must be mapped to the command function using the `map` command; see the `map` command for a list of valid values for *keyname*. The keyname is abbreviated to reduce the size of the menu. For example, `-key control+Z` is displayed as `Ctrl+Z`.

If no commands are assigned to the menu item with the `-cmd` option, then the menu item executes whatever is mapped to *keyname*.

`-keytext` "*text*" adds the accelerator key text displayed in the right-hand field of the menu item. If "*text*" is `none`, then no accelerator is shown. The `-keytext` option is similar to the `-key` option except that the accelerator text is specified directly by *text* instead of by a keyname. Since the `-keytext` option requires more memory, the `-key` option is preferred. However, `-keytext` is useful for turning off accelerator text (using `none`) and also for menu items mapped to multiple keys, for example, using prefix keymaps, since `-key` only allows a single key to be specified.

`-toggle` '*togexpr*' specifies that the menu item is a toggle item (that is, it displays a check mark if set). The string argument *togexpr* specifies an ACL expression to be evaluated when the menu is posted. If it evaluates to non-zero the menu item will be displayed with a check mark, otherwise, the check mark will be removed.

togexpr can also be surrounded with double quotes.

`-radio` specifies that the toggle menu item is part of a radio group, that is, it is one a set of mutually exclusive items. All adjacent toggle-type items with the `-radio` option are considered part of the group. Depending on the window system, a different icon is displayed for radio group items.

`-active` "*actexpr*" specifies an expression to be evaluated when the menu is posted that determines whether the menu item is enabled or disabled (grayed-out). If the expression evaluates to non-zero the menu item is selectable, otherwise, the menu item is grayed-out.

`-separator` specifies that this menu item is a separator line. In this case *label* should be a null string and no additional options are allowed.

To save menu additions across sessions, you must either write out your changes with the `menu_save` command (this creates a `menu.cf` file in the document directory which can be loaded automatically when the document is opened), or you can put your mappings in the *docname*`.acl` file for the document.

is a synonym for `menu_add`.

Examples

```
mad .File. "Make all caps" -cmd {tra uc}
mad -before .File.Make_all_caps "Capitalize first" \
 -cmd {tra pc}
menu_add .Options. "" -separator
menu_add .Options. "Find &Expressions" \
 -toggle 'option("expr")=="on"' \
 -cmd {if (option("expr")=="on") {set expr=off;} \
 else {set expr=on;}}
```

menu_add -menu

`menu_add -menu` [`-before`] *destination label*

This command adds a menu to the menu bar. *label* is the name for the new menu and *destination* is a menu path indicating the existing menu that the new menu will follow. If `-before` is specified, the new menu will appear before the destination menu rather than after it.

Note

Underscores, spaces, and periods are not allowed in menu bar labels.

To add a user-defined shortcut menu, you must specify the *destination* as `:` (colon) *label*. You must then add menu items to the shortcut menu, and post it using the `menu_popup` function on page 415.

To add items to your new menu, use the `menu_add` command on page 674 without the `-menu` option.

The following example adds the **Test** menu to the menu bar after the **Tools** menu:

```
menu_add -menu .Tools "Test"
```

The abbreviation of `menu_add` is `mad`.

The following example adds the **Test** menu to the menu bar before the **Find** menu:

```
mad -menu -before .Find "Test"
```

menu_change

`menu_change` *path* [`-label` “*name*”] [`-cmd` {*cmds*}] [`-key` *keyname* | `-keytext` [“*text*” | none | default]] [`-active` [“*actexpr*”] | `-inactive`] [`-toggle` '*togexpr*']

This command changes an existing menu item. is a synonym for `menu_change`.

- *path* is the menu path of the menu item to be changed.
- `-label` "*name*" changes the label displayed for the item. If name contains any variable references, for example, `-label "Modify $tagname"`, the variable reference is substituted into the label string each time the menu containing the item is posted. Note, the label string *name* must be enclosed in single quotes to prevent the command parser from expanding the variable reference when the command is parsed.

> **Note**
> Underscores, spaces, and periods are not allowed in menu bar labels.

- `-cmd {`*cmds*`}` changes the command(s) associated with the menu item. The command must be enclosed in curly braces. Multiple commands are separated by semicolons.
- `-key` *keyname* changes the accelerator key text displayed in the right-hand field of the menu item. See `map` for a list of possible values for *keyname*. *keyname* is abbreviated to reduce the size of the menu. For example, `-key control-z` is displayed as `Ctrl+Z`.

> **Note**
> Specifying a *keyname* does not actually map the specified key combination to the menu item. You must use the `map` on page 669 command to associate a user function to a key.

- `-keytext` has three options, `-keytext "`*text*`"`, `-keytext none` and `-keytext default`.
  - `-keytext "`*text*`"` changes the accelerator key text displayed in the right-hand field of the menu item
  - `-keytext none` means no accelerator is shown.
  - `-keytext default` restores the default keyboard accelerator. You may want to use it following a `menu_change -keytext none` command.

  The `-keytext` option is similar to the `-key` option except that the accelerator text is specified directly by *text* instead of by a keyname. Because the `-keytext` option requires more memory, the `-key` option is preferred. However, `-keytext` is useful for turning off accelerator text (using `none`) and also for menu items mapped to multiple keys, for example, using prefix keymaps, since `-key` only allows a single key to be specified.

- `-active` ['*actexpr*'] specifies an expression to be evaluated when the menu is posted that determines whether the menu item is enabled or disabled (grayed-out). If the expression evaluates to non-zero the menu item is selectable, otherwise, the menu item is unavailable. If *actexpr* is omitted, the menu item is made active unconditionally.
- `-inactive` specifies that the menu item be made insensitive (grayed-out). The item remains unavailable until a subsequent `menu_change` command enables it with the `-active` option. If the menu item is enabled using the `-active` option, it is still subject to be unavailable when posted (according to the context of the cursor). This context-sensitive graying is done only for menu items mapped to simple commands, such as `delete_mark`, `insert_tag`, `undo`. The optional string argument *actexpr* specifies an expression to be evaluated when the menu is posted. That determines whether the menu item is enabled or disabled (grayed-out). If the expression evaluates to non-zero, the menu item is selectable, otherwise, the menu item is grayed-out.
- `-toggle` '*togexpr*' changes the `-cmd` menu item into a toggle item, which displays a check mark if set. The string argument *togexpr* specifies an expression to be evaluated when the menu is posted. If it evaluates to non-zero the menu item will be displayed with a check mark, otherwise, the check mark will be removed.

Examples

```
menu_change .File.Print -inactive
mch :EditPopup.Paste -key control-y
mch ".Options.Full Menus" -check
mch ".File.Preview.Current*" -label " to end"
```

menu_copy

`menu_copy` *item* [`-before`] *destination*

This command copies an item on a menu from one location to another. *item* is a menu path indicating the item to be copied and *destination* is a menu path indicating the existing item that the new item should follow. If `-before` is specified, the new item appears before the destination item rather than after it.

is a synonym for `menu_copy`.

Examples

```
menu_copy .File.Save .Quick
mcp Insert_Table -before .Quick.Save
```

menu_delete

`menu_delete` *item*

This command deletes a menu from the menu bar or deletes an item from an existing menu. *item* is a menu path indicating the menu or item to be deleted.

is a synonym for `menu_delete`.

Examples

```
menu_delete .Quick.Save
mdel .Quick
```

menu_load

`menu_load` [`-d` *name* [`-d` *name*]] [`-u` *name* [`-u` *name*]] [*filename*]

A `menu.cf` file is a menu configuration file created and placed in the directory containing the document file when you use the `menu_save` command. `menu_load` loads the specified menu configuration file (*filename*). If no menu file is specified with `menu_load`, Arbortext Editor determines the menu configuration as follows.

- The `menu.cf` file in the directory containing the document file is used.
- If there is no `menu.cf` file in the directory containing the document file, then the `menu.cf` file in the directory specified by the environment variable *$APTMENUFILE* is loaded.
- If `menu.cf` is not found in the path and directory specified by *$APTMENUFILE*, the default menu configuration file located at *`Arbortext-path`*`\lib\editmenu.cf` is loaded.

Therefore, to reinstate your default menu after loading a custom menu, use `menu_load` without specifying a menu file.

The `-d` option with `menu_load` defines the associated symbol *name* for the processor used to parse the `menu.cf` file.

The `-u` option with `menu_load` removes the definition of the associated symbol *name* for the processor used to parse the `menu.cf` file.

More than one *name* can be defined (or have its definition removed) by specifying multiple `-d` (or `-u`) options.

Examples

```
menu_load menu.cf
menu_load /mdm/book.men
menu_load -u fullmenus
```

menu_move

`menu_move` *item* [`-before`] *destination*

This command moves a menu or an item on a menu from one location to another. *item* is a menu path indicating the menu or item to be moved and *destination* is a menu path indicating the existing item that the new menu or item should follow. If `-before` is specified, the new item appears before the destination item rather than after it.

is a synonym for `menu_move`.

Examples

```
menu_move .File.New .File.Preview
mmv .File -before .Help
```

menu_reset

`menu_reset`

This command resets all menus to the state they were when the menus were loaded. That is, if you have changed your current menus using `menu_add`, `menu_delete`, `menu_move`, or another menu modification command, the `menu_reset` will undo those changes.

Examples

```
menu_reset
```

menu_save

`menu_save` [`-all`] [`-buttons`] [`-popups`] [*pathname*]

This command saves your current menu configuration in a file called `menu.cf` in the directory containing the document or to the path name that you specify. By default, **Help** and **File** menus are not saved; the `-all` option causes these menus to be saved as well.

`-buttons` specifies that the menus defining the menu bar buttons for the current window are to be saved. This is the default.

`-popups` specifies that the shortcut menus are to be saved. By default, only user-defined or changed built-in menus are saved. If `-all` is given, then all shortcut menus are saved.

To save the menu bar menus and the shortcut menus, specify both `-buttons` and `-popups`.

Examples

```
menu_save
menu_save -all /ati/doc/master.men
menu_save -all -popups popup.men
```

message

`message` "*text* " ["*text* " …]

This command generates a message window containing the specified text. Either single or double quotes can be used to mark the text. Every set of quotation marks contains a new line of the message. If the `message` command is read in from a command file, a line break may be introduced by putting a backslash (\) at the end of a line.

If the current window has a message area at the bottom of the window and the message is short enough to fit, it is displayed in the message area instead of a separate window.

Examples

```
message "sending print job to auxiliary printer"
mes "WARNING:" "This report is not yet approved."
mes 'Substitute command successful \
 "QAP" changed to "Quality Assurance Program" \
 in all cases'
mes "You are currently editing $docname."
```

mkdir

`mkdir` [`-p`] *pathname*

This command creates a new directory specified by *pathname*. If `-p` is not included, all leading components in *pathname* must be existing directories, and only the lowest level directory will be created. When the `-p` option is given, `mkdir` will create a multiple-level directory structure.

Examples

```
mkdir techman
mkdir ../src
# The following command creates the entire directory structure.
mkdir -p 'c:\newdir\tests\monday'
```

modify_entity

`modify_entity` [*name*]

This alias opens a dialog box for any text, file, or graphic entity that has been declared in Arbortext Editor, allowing modification of its definition. If the entity name is not supplied, you are prompted for it.

is a synonym for `modify_entity`.

Examples

```
modify_entity
me ati
```

modify_file_entities

`modify_file_entities`

This alias brings up the file entities panel which displays name, number of uses, and system and public identifiers for all file entities (including those declared in the DTD).

You can use this panel to rename any file entities declared in the private markup section of the document instance, to locate instances of file entity use, and to modify system and public identifiers for entities declared in the private markup section.

is a synonym for `modify_file_entities`.

Examples

```
modify_file_entities
mfe
```

modify_graphic_entities

`modify_graphic_entities`

This alias brings up the graphic entities panel which displays name, number of uses, and system and public identifiers and notations for all graphic entities (including those declared in the DTD).

You can use this panel to rename any graphic entities declared in the private markup section of the document instance, to locate instances of graphic entity use, and to modify system and public identifiers and notations for entities declared in the private markup section.

`mge` is a synonym for `modify_graphic_entities`.

Examples

```
modify_graphic_entities
mge
```

modify_marked_section

`modify_marked_section` [*status1* [*status2* …]]

This command changes the status of the marked section before the cursor to the status specified, prompting for the new status if necessary. Status keywords may be references to declared marked section parameters or may be marked section keywords.

is a synonym for `modify_marked_section`.

Examples

```
modify_marked_section _ignore
mms %teachers
```

```
mms %trainers TEMP
mms %cobol INCLUDE CDATA
```

modify_ms_parameters

`modify_ms_parameters` [*parametername1 status1* [*parametername2 status2* …]]

If no parameters are given, this command brings up the **Marked Section Parameters** panel which displays name, number of uses, and status information for all marked sections declared in the private markup section of the document instance.

Otherwise, it sets *parametername1* to *status1*, *parametername2* to *status2*, and so forth. In order for this operation to work, the document must remain in context after all affected marked sections are changed. Otherwise, no changes will be made. Also, the document must be saved before you can apply changes to a marked section.

You can also use this panel to rename marked sections, to locate instances of use, and to modify status information.

is a synonym for `modify_ms_parameters`.

Examples

```
modify_ms_parameters
mmsp
```

modify_notation

`modify_notation` *notationname*

This alias allows you to modify the declaration of the specified notation, prompting for a notation name if none is supplied.

is a synonym for `modify_notation`.

Examples

```
modify_notation fax
```

modify_notations

`modify_notations`

This alias brings up the notations panel which displays the name, number of uses, and system and public identifiers for all notations declared in the private markup section of the document instance. You can also use this panel to rename notations, locate instances of use, and modify system and public identifiers.

is a synonym for `modify_notations`.

Examples
`modify_notations`
`mns`

modify_tag

`modify_tag` [`-local` [*tagname*] | `-global` [*tagname*] | `-quick` | `-quickattr`] { [[*attr*=*value*] …] | [`-inline`] [`-modal`] [`-attr` *attrname*]}

In the Edit view, this command opens the **Modify Attributes** dialog box for modifications to either the tag or entity reference immediately before the cursor, or the specified tag or entity reference. In the Document Map view, this command applies to the tag to the right of the cursor if the cursor is at the start of a line (that is, between the element icon and the element name); you can control this behavior with the `set docmapcurrenttag=off` option.

For tags defined by your DTD, local modifications can be made to SGML/XML attributes. Unless you are modifying a user-defined tag, attributes can only be modified locally. Tags supplied by Arbortext Editor, such as `_link`, have attributes that can be modified with this command.

If `-local` is specified (the default), only the current instance of the tag will be affected. Attributes that are set locally override attributes that are set globally.

The `-global` option is generally able to modify non-DTD markup such as user-defined file and text entities, marked sections, and processing instructions. It cannot modify attributes or elements of a DTD. Where `-global` is allowed and specified, it modifies every occurrence of the tag in the entire document.

The `-quick` (or `-quickattr`) option will launch the **Quick Attribute** dialog box, where you can modify a single attribute for the specified element.

The `-inline` option causes the Column view attribute value cell with focus, if any, to be opened for modification. The command fails silently if the current view is not a Column view or if no cell has focus.

If `-modal` is specified, the **Modify Attributes** dialog box is launched as a modal dialog box.

The *attr=value* option modifies attributes without opening the **Modify Attributes** dialog box. Up to 50 *attr=value* pairs can be specified with a single command. You can determine the names of attributes from SGML/XML files.

Note

If you have applied an Alias Map Overview to the document, *attr* and *attrname* can be either aliases or real names.

The *attr=value* option can be used to set invalid attributes to new values. New invalid attributes cannot be added using this technique.

In a Free-form document type instance, the *attr=value* option can be used to alter the value of existing attributes and to add new attributes and values. New attribute names must be valid XML names.

If the `-attr` *attrname* option is specified, then the keyboard focus for the **Modify Attributes** dialog box is set to the value field for the specified attribute name. By default, the focus is set to the first attribute field.

modify_tag Command Options

Where options may be used		
Markup tags	**-global**	**-local**
DTD element tags	no	yes
User-defined tags of DTD tags	no	yes
`_comment` tags	yes	no
File and text entity tags	yes	no
Symbol tags	no	no
Marked sections tags	no	no
Formatting processing instructions: `_font`, `_kern`, `_texmac`, `_texmacpair`	no	yes
Other processing instructions such as `_link`	yes	yes
User-defined tags of processing instructions	yes	yes
Exceptions for FOSIs: `charlist` within a `docdesc`	no	no
Help documents: `document` tag	yes	no

is a synonym for `modify_tag`.

Examples

```
modify_tag ID='org-chart'
mt -local
mt -local par ParaType="block"
mt -global -caret
mt -global chaphead OverallTypeSize=18pt \
Underl=true UnderlRuleWidth=width-of-heading
```

modify_text_entities

`modify_text_entities`

This alias brings up the **Text Entities** dialog box, which displays the name, number of uses, and system and public identifiers for all text entities (including those declared in the DTD).

You can use this dialog box to rename any text entities declared in the private markup section of the document instance, to locate instances of text entity use, and to modify system and public identifiers for entities declared in the private markup section.

is a synonym for `modify_text_entities`.

Examples

```
modify_text_entities
mte
```

move_file

`move_file` { *oldname newname* | *file1* [*file2* …] *dir*}

This command moves the file *oldname* to *newname*. This command can be used to rename a file or a document.

The second form of `move_file`, `move_file` *`file1`* [*`file2`*] `...`*`dir`*, moves one or more file names (which may also be directories) preserving their original names, into the last directory in the list.

`move_file` cannot be used to move a directory across file systems. The `copy_file` and `remove_file` commands must be used instead.

is a synonym for `move_file`.

Examples

```
move_file drugs.txt /apr3bust
mv /usr/svc/newslet/list.sgm /tecwrite/maillist
mv demo.sgm demo.acl ../article
```

ms_hide

`ms_hide`

This alias hides ignored marked sections in a document.

Example

```
ms_hide
```

ms_show

`ms_show`

This alias displays ignored marked sections in a document.

Example

```
ms_show
```

new

`new` [`-stylesheet` *name*] [`-newwindow` | `-ok`] [`-type` *doctype*] [`-sample`] [*docname*]

This command replaces the document in the Edit pane with a newly created document.

`-stylesheet` *name* specifies the stylesheet to be used in place of the default stylesheet for the Edit view. *name* is the path and file name of a stylesheet file. *name* can also be the base name of the stylesheet file (the file name without the path). If just the base name is supplied, Arbortext Editor looks for the file in the document directory, and then the document type directory.

`-newwindow` specifies that a new window be created to display the new document. If `-newwindow` is not specified, the new document replaces the current document. is a synonym for `-newwindow`.

`-ok` moves you to the requested document without prompting you about unsaved changes to the current document or file (unsaved changes will be lost). This option is not applicable if `-newwindow` is given.

`-type` creates a new document of the specified document type. The type can be an initial substring from the list of document types listed in the **New Document** dialog box. If the name contains blanks, the name must be enclosed in quotes (for example “personal letter”). Any new document type installed as a template is also legal. This option can be combined with the `-sample` option. If `-type` is not specified, the **New Document** dialog box is opened.

`-sample` specifies that the new document is to contain sample markup.

docname specifies the path name to use when the document is saved. If omitted, the **Save As** dialog box will prompt for the name when saving.

Examples

```
new -sample book.sgm
new -type memo -nw salesmtg.sgm
```

newline

`newline`

This command generates a return.

is a synonym for `newline`.

Examples

```
newline
nl
```

options

`options` [[*categoryname*]]

This command brings up the **Preferences** dialog box from which various options affecting the editing environment can be set. Enter a value for *categoryname* to automatically bring that category of the dialog box to the foreground. The category names are:

- `Advanced` invokes the **Advanced Preferences** window, for editing set command options
- `Colors` corresponds to the Colors category
- `Columns` corresponds to the Columns category
- `Compare` corresponds to the Compare category
- `DITA` corresponds to the DITA category
- `Edit` corresponds to the Edit category
- `File` corresponds to the File Locations category
- `PE` corresponds to the Publishing Engine category
- `Save` corresponds to the Save category
- `Spell` corresponds to the Spelling category
- `User` corresponds to the User Information category
- `View` corresponds to the View category
- `Warnings` corresponds to the Warnings category
- `Window` corresponds to the Window category

Note that all of these options can also be set using the `set` on page 744 command.

Examples

```
opt
options Window
Options window
```

outline

`outline` [div | backup]

This command inserts new division tags of the requested type and moves the cursor within those tags when:

- `outline div` (the default) starts a new division of the same type after the end of the current division.
- `outline backup` moves the cursor past the end tag of the current division so that the next `outline div` command will start a division one level higher than the current division.

Examples

```
outline
{outline backup; outline div}
```

package

`package` [*packagename*]

This command declares the name of a package when used at the top of a command file. Packages let you declare global variables and functions whose name space is localized to one or more command files.

Example

```
package myapp
```

paste

`paste` [*buffername*] [`-after`]

This command inserts the text from the specified paste buffer at the current cursor location. In the Edit view, the cursor is placed after the pasted region. In the Document Map view, the cursor is placed before the pasted region; you can control this behavior with the `set docmappastecaret=off` option.

If a buffer name is given, that buffer is used. Otherwise, text is inserted from the current buffer (see `set paste=`*`buffername`*). Once text is placed in the paste buffer, it can be pasted back any number of times (until it is replaced).

The optional `-after` flag is used when pasting table objects. If `-after` is specified, the table object(s) in the paste buffer will be pasted after the cell, row, or column containing the text caret. If `-after` is omitted, the table object(s) in the paste buffer will be pasted before the cell, row, or column containing the text caret.

Examples

```
paste
pas bufA
pas seealso
pas tablecells -after
```

preview

`preview` [onepass | allpasses | noformat | noprompt | quit | stop] [auto | force] [wait | nowait]

This command formats the current document and displays it in the **Print Preview** window. The abbreviation `pre` is a valid substitute for `preview`.

- `onepass` — (The default.) Displays the document in the **Print Preview** window after performing a single formatting pass. If one pass is not sufficient for final output, then the Arbortext Editor status bar will indicate `re-fmt needed` and another preview command will trigger another pass.

> **Note**
>
> The `onepass` option is the default for the `preview` command, but `allpasses` is the default for the `print` command on page 691.

- `allpasses` — Displays the document in the **Print Preview** window after performing as many formatting passes as necessary for final output. Arbortext Editor performs one pass if there are no page number references in the document, or if the value of the page numbers can be correctly inferred from a previous formatting operation during the current Arbortext Editor session.
- `noformat` — Displays the document in the **Print Preview** window using the existing formatting data file (`.dvi` file) for the document (if it exists). Use this option carefully. It is useful if you need draft output for a large document in a hurry and do not want to take the time to reformat minor changes you have made. It could also be used in carefully controlled applications to format in one Arbortext Editor session, and print or preview in another Arbortext Editor without the need to reformat the document.
  - If formatting was done in a previous Arbortext Editor session, then Arbortext Editor's ability to match the display in the **Print Preview** window to the cursor location in the Edit view is disabled until a formatting pass has been done in the current Arbortext Editor session.
  - The `force` and `auto` arguments have no effect when `noformat` is specified.
  - The `preview noformat` command does not affect the document's formatting status (that is, `fmt-needed`, `refmt-needed`).
- `noprompt` — Suppresses the **Print Preview** window. This option is ignored when publishing using Arbortext Publishing Engine.

  When `noprompt` is in effect, the print stylesheet (as specified by the `printstylesheet` Advanced Preference and `set` option) is used to generate the preview of the document. If no print stylesheet is set, the edit stylesheet is used.

  Independent of this option, the **Print Preview** window is suppressed when publishing without using Arbortext Publishing Engine when the document's

document type has its *AllowComposeStylesheetlist* attribute set to `no` in the `.dcf` file.

- `quit` — Closes the **Print Preview** window and continues formatting.
- `stop` — Halts document formatting and leaves the **Print Preview** window open.
- `auto` — (The default.) Lets Arbortext Editor decide whether the document needs to be formatted or not before being previewed. Arbortext Editor will trigger formatting if the document has not been formatted yet during the current session, or if the previous formatting command left the document in the `refmt-needed` state, or if the document has been modified since the preceding formatting command. If the document is up to date with regard to formatting, then `preview auto` will cause Arbortext Editor to position the Print Preview window to the page containing the cursor.
- `force` — Performs at least one formatting pass regardless of whether Arbortext Editor thinks it is necessary.
- `wait` — Makes editing unavailable in Arbortext Editor until a preview request completes.
- `nowait` — (The default.) Allows editing in Arbortext Editor while a preview request is being processed.

Examples

```
preview (invokes preview onepass auto)
pre onepass
pre onepass force
pre allpasses auto
pre stop
pre quit
```

print

`print` editor [panel] [all | screen | selection] [portrait | landscape] [header= { pageno | datemark | none | '*string*'}] [footer= { pageno | datemark | none | '*string*'}] [leftmargin= *n*] [topmargin= *n*] [printer= *printer_name*] [file= *path_name*] [color | monochrome] [copies=*n*]

and

`print` composed [panel] [all | current | *page_range* | *page_list*] [onepass | allpasses | noformat] [force | auto] [wait | nowait] [portrait | landscape] [collate | nocollate] [papersize= { uslegal | usletter | a4 | b5}] [paperheight=*n*] [paperwidth=*n*] [printer=*printer_name*] [file=*path_name*] [color | monochrome] [copies=*n*] [stylesheet=e3:*n*]

The `print` command prints the specified portion of the document. There are two main arguments to this command: `composed` and `editor`.

Note

If you do not specify the `editor` or `composed` arguments, the `print` command defaults to the `composed` argument if the Print Composer is installed and you are printing a non-ASCII document. It defaults to the `editor` argument if the Print Composer is not installed or you are printing an ASCII document.

`editor` Argument Options

Use the `editor` argument to print the document as it displays in the Edit view, Document Map, or Column view (whichever has focus). The output will reflect the current display font size and tag display settings.

The following options are valid when the `editor` argument is valid.

- `panel` — Opens the **Print Editor View** dialog box for printing.

  You can make the selections for printing using the dialog's controls instead of argument options, if required.
- `all`, `screen`, and `selection` — Control what portion of the document instance to print:
  - `all` (the default) — Prints the entire document.
  - `screen` — Prints the portion of the document that is currently in the Edit pane.
  - `selection` — Prints only the highlighted portion of the document.
- `portrait` and `landscape` — Control the orientation of the output on the paper. If neither is specified, `portrait` is assumed.
- `header` and `footer` — Specify what will print on the top and bottom of the page. The following options are associated with the `header` and `footer` arguments:
  - `pageno` (the default footer) — Prints the ordinal page number.
  - `datemark` (the default header) — Prints a datemark that identifies the document name, date, time, and user ID for the job.
  - `none` — Suppresses the printing of headers or footers.
  - `'string'` — Allows you to specify a string delimited by a pair of single quotes (') or double quotes (") to include in the header or footer.
- `leftmargin` and `topmargin` — Allow you to specify in decimal numbers the indentation from the left or top edge of the paper, respectively, to the point

at which the text area is to begin. If you do not specify the unit on the indentation, inches are assumed.

- `printer` — Sends the document to a specific printer. If the name includes spaces or special characters, the name must be enclosed with quotes.
- `file` — Writes a PostScript version of your document to the specified file.
- `color` — Generates color PostScript (requires PostScript level II compatible printer) or `monochrome` for black and white. `Color` is the default.
- `copies` — Specifies the number of copies to be printed.

Following are examples of the `print editor` command:

```
print editor all
print editor screen
print editor selection
print editor selection \
 header='FORMAT ERROR' footer=datemark \
 leftmargin=2
print editor screen header=none \
 footer=pageno topmargin=2.5cm
print editor selection printer=lw \
```

`composed` Argument Options

The `composed` argument outputs a formatted copy of your document. The following options are valid when the `composed` argument is valid.

- `panel` — Opens the **Print Published View** dialog box for printing.

  You can make the selections for printing using the dialog's controls instead of argument options, if required.
- `all`, `current`, *page_range*, *page_list*— Control what portion of the document to print:
  - `all` (the default) — Represents the entire document; this option presumes forward page ordering and will start with page one. If you wish to print an entire document in reverse order, use the *page_range* option or print from the **Print** panel, selecting **Reverse** for page ordering along with Pages to print: **All**.
  - `current` — Represents the formatted page from the currently active Arbortext Editor window. If the **Print Preview** window is active, the current page is the page(s) displayed in the **Print Preview** window. If the **Edit** window is the active window, Arbortext Editor prints the page on which the cursor is located when it is formatted.

- *page_range* — Represents the ordinal page numbers for the portion of the document to be printed. (If your document has four pages of preface material, the page with the number "2" on it may actually be ordinal page 6.) You can specify a single number (for example, `2` prints ordinal page 2); a range of pages going from either smaller to larger or larger to smaller (for example, `5-3` prints ordinal pages 5, 4, and 3, in that order); or a series of pages and page ranges (for example, `1,3-5,7` prints ordinal pages 1, 3, 4, 5, and 7).

  If you wish to print an entire document in reverse order, reverse the pages for the *page_range* option so that the page number of the last page of the document is on the left-hand side of the page range separator (for example, `87-1`). You will need to know the exact number of pages in your document. You may want to print in reverse order if your printer stacks the first page at the bottom of the pile. (Note that some documents that include PostScript prologues may not print with reverse page ordering.)

  Valid page specification characters are `0 1 2 3 4 5 6 7 8 9`. The separator characters are `-` (a page range separator), `:` (a page count separator), and `,` (a token separator). A page range of the form *`pg:count`* starts with the page of the first number and prints the number of pages specified by the second number. A `,` separator can separate a series of page range specifications in any of the supported formats. For example:

  ```
  3-5 specifies pages 3, 4, and 5
  1,4,7-6 specifies pages 1, 4, 7, and 6
  2:3 specifies pages 2, 3, and 4
  4:-3 specifies pages 4, 3, and 2
  ```

  If you are specifying pages in reverse order, the absolute value of a negative page number can't exceed the starting page number (making a specification such as `4:-5` invalid). All white space (blank, tab, newline, CR, LF) is ignored, with the exceptions that newline, end of line, or null character are interpreted as the end of a token.

- *page_list* — is a text file containing a list of page specifications. The page specification list allows the same formats supported by the *page_range* parameter with the following extensions:

  - A new line (as well as a comma) can be used to separate page specifications.
  - A page range specification must be on one line. Do not put the first page of the range and a hyphen on one line and then the last page of the range on the next line.

  For example:

  ```
  1,8-9,12
  37-55
  ```

```
87-237
259:45
```

specifies pages 1, 8, 9, 12, 37 through 55, 87 through 237, and 259 through 303.

- `onepass`, `allpasses`, and `noformat` — Control the formatting process that precedes the print:
  - `onepass` — Prints the document after performing a single formatting pass. If one pass is not sufficient for final output, then the Arbortext Editor status bar will indicate `re-fmt needed` and another `print` command will trigger another pass.
  - `allpasses` (the default) — Prints the document after performing as many formatting passes as necessary for final output. Arbortext Editor performs one pass if there are no page number references in the document, if pass reduction is enabled and succeeds, or if the value of the page numbers can be correctly inferred from a previous formatting session during the current Arbortext Editor session. Otherwise, Arbortext Editor performs multiple passes.

    Note

    For compatibility purposes, the obsolete `final` option for the `print` command (used for published output in releases preceding Epic Editor 4.0) is mapped to the `allpasses` option.

  - `noformat` — Prints the document using the existing formatting data file (`.dvi` file) for the document (if it exists). Use this option carefully. It is useful if you need draft output for a large document in a hurry and do not want to take the time to reformat minor changes you have made. It could also be used in carefully controlled applications to format in one Arbortext Editor session, and print or preview in another Arbortext Editor without the need to reformat the document.

    Note

    Printing with the `noformat` argument does not affect the document's formatting status (that is, `fmt-needed`, `refmt-needed`).

- `auto` and `force` — Control how Arbortext Editor decides whether a document instance requires formatting:
  - `auto` (the default) — Lets Arbortext Editor decide whether the document needs to be formatted or not before being printed. Arbortext Editor will trigger formatting if the document has not been formatted during the current session, or if the previous formatting command left the document in the `refmt-needed` state, or if the document has been modified since the preceding formatting command. If the document is up to date with regard to formatting, then `preview auto` will print the specified portion of the document without reformatting it.
  - `force` — Forces at least one formatting pass regardless of whether Arbortext Editor thinks it is necessary.

> **Note**
>
> `force` and `auto` have no effect when `noformat` is specified.

- `wait` and `nowait` — Control whether Arbortext Editor should wait for the print to complete (that is, spool) before continuing to execute commands:
  - `wait` — Instructs Arbortext Editor to wait until the format process is complete (that is, spooled) before continuing with subsequent commands. If formatting fails, your function can return a non-zero error status and retrieve the error message from the variable *main::ERROR*.
  - `nowait` — Instructs Arbortext Editor to continue processing and perform the print in the background. This is what usually happens when you select **File ► Print Preview** or **File ► Print Published**.

> **Note**
>
> The `nowait` option is the default. However, if you are running Arbortext Editor with a `-c` startup option from the shell, Arbortext Editor will always wait for the print to complete, even if you specify the `nowait` option.

- `portrait` and `landscape` — Control the orientation of the output on the paper. If neither is specified, `portrait` is assumed.
- `collate` and `nocollate` — Controls whether the output is collated. If neither is specified, `nocollate` is assumed.

- `papersize` — Controls the size of the paper on which the document is printed. Specify one of the following options:
  - `uslegal`
  - `usletter`
  - `a4`
  - `b5`

> **Note**
>
> If you do not specify an option, `papersize` defaults to the current printer's paper size setting.

- `paperheight` and `paperwidth` — Allow you to specify the height and width of nonstandard paper. Allowable units of measure are: `pc`, `pt`, `cm`, `mm`, and `in`.
- `printer` — Sends the document to a specific printer if you have more than one printer attached to your system.
- `file` — Writes a PostScript version of your document to the specified file.
- `color` — Generates color PostScript (requires PostScript level II compatible printer) or `monochrome` for black and white. `Color` is the default.
- `copies` — Specifies the number of copies to be printed.
- `stylesheet` — Specifies a stylesheet for a publishing request to the Arbortext Publishing Engine from Arbortext Editor. You must specify `e3:`*`stylesheet-path`*, where the path is an absolute path to the stylesheet file name. This option is not available if you use `panel`. The option is ignored if you are not using the Arbortext Publishing Engine for publishing documents.

Following are examples of the `print composed` command:

```
print composed
print composed current
print composed 3
print composed 2-6
print composed 14-11,9,7-3,1
print composed all copies=4 printer=lw
print composed file=thesis
```

quit

`quit` [ok]

This command closes the current window. If you only have one window open, this command exits from Arbortext Editor without saving the document. If there are unsaved changes in your document, a panel informs you and asks for permission to exit without saving the changes. You can bypass the panel by typing `quit ok`.

Examples

```
quit
qui ok
```

read (Command)

`read` [{ `-paste` | `-buffer` *buffername*} [`-append`]] [`-sgml` | `-untaggedascii` | [`-xml`] [`-encoding` *string*] [`-cc` | `-nocc`] [`-pendingdelete` | `-nopendingdelete`] *filename*

This command inserts the text of the requested file at the point of the cursor.

`-paste` loads the text of the requested file into either the current or a named paste buffer.

`-buffer` *buffername* forces the information that is read to be loaded into the named buffer specified by *buffername*. If the buffer already exists it will be overwritten, else it is created.

`-append` adds text to the end of the buffer file rather than replaces the contents of the buffer.

`-sgml` allows you to read in an SGML document that does not have the SGML header created by the `write -sgml` command, that is, starting with `<!DOCTYPE`.

`-untaggedascii` allows you to read in the named document as ASCII. Tagging, if any, are interpreted as literal text, not Arbortext Editor tags. `-untagged` is a synonym for `-untaggedascii`.

`-xml` reads an XML document into the current document (SGML or XML). When this option is selected, a completeness check will not be performed (that is, `-nocc` is implied), and specifying `-cc` will have no effect.

`-encoding` determines the encoding of the file being read, as specified by *string*; *string* may be one of the following strings:

Supported Encodings

Adobe-Standard-Encoding	Shift_JIS
CNS11643	Big5
ISO-10646-UCS-2	GB2312
ISO-8859-1 to ISO-8859-11	KSC_5601
ISO-8859-13 to ISO-8859-16	EUC-JP

Supported Encodings (continued)

windows-1250 to windows-1258	CEUC
US-ASCII	TEUC
UTF-16	EUC-KR
UTF-8	cp932
cp949	cp936
cp950	

`-cc` forces Arbortext Editor to do a completeness check when it opens a file created with Arbortext Editor. (By default, a completeness check is not done if the file being read in was saved in Arbortext Editor.) This option is recommended when opening documents which were created in Arbortext Editor but were later edited elsewhere and that may no longer be normalized or complete.

`-nocc` does not allow Arbortext Editor to do a completeness check when reading a document or file. (By default, a completeness check is done after the file is read in.) In this case, the document should be fully normalized.

`-pendingdelete` replaces any selected text in the document with the file being read.

`-nopendingdelete` preserves any selected text in the document and places the file being read to the right of the current selection.

Note

If neither `-pendingdelete` or `-nopendingdelete` are set, the command replaces any selected text according to the current setting of the `set pendingdelete` on page 866 command.

filename is the file name and path (if needed).

Examples

```
read mydoc
rea -buffer buf1 wishlist.sgm
rea -sgml /lxr/grants89/proposal.sgm
rea -untag budget.sgm
rea -xml http://www.some_site.com/examples/doc1.xml
```

readvar

`readvar` [`-help` "*text*"] [`-title` "*string*"] [[`-prompt` "*string*"] [`-value` "*string*" | `-choice` "*c1|c2|c3…cN*" [`-default` "*string*"]] *variable1*] [[`-prompt` "*string*"] [`-value` "*string*" | `-choice` "*c1|c2|c3…cN*" [`-default` "*string*"]] *variable2*] …

The `readvar` command prompts for the values of the variables named. The information you provide is used to assign values to those variables. (Note that the variable is created if it does not already exist).

The `readvar` command provides a dialog box that contains **OK** and **CANCEL** buttons. If you click on **OK**, `main::status` is set to `0`. If you click on **CANCEL**, **main::status** is set to `1`.

`-help` "*text*" specifies that a **Help** button should be added to the panel and, if pressed, the help text *text* will display in a Help window.

`-title` "*string*" specifies a string to be used as the window title for the readvar dialog box.

`-prompt` identifies a string to be supplied as the text of the prompt. Multiple prompts on the same line are allowed.

`-value` allows you to specify a string value at the prompt and use it as a default text entry value.

`-choice` specifies a list of mutually exclusive choices (radio buttons). You can stack the radio buttons vertically on the panel by starting the choice string with `|` (that is, the first field is empty). If you specify a single item choice list, the choice will appear as a check box instead of a radio button. If the check box is selected, the value is set to a one (`1`). If the check box is cleared, the value is set to a zero (`0`).

Note

The `-choice` string will be truncated at 49 characters. If you want to specify a `-choice` string longer than 49 characters, use the `list_response` function on page 403.

`-default` allows you to specify a choice string as the default value. It can only be used with the `-choice` option. Only one of the options `-value` and `-choice` may be specified for a given variable name.

variable is a name, which should be supplied without the leading dollar sign. Variable names are limited to 199 characters each.

To establish a mnemonic (hot key) for a particular dialog box item, precede the mnemonic letter with the '&' character in the associated string. Use "&&" anyplace you want a single '&' to show.

You can prompt for up to 20 variables on the same panel.

Examples

```
readvar docname
readvar -prompt '..............' -prompt 'Enter story name:' story
readvar -prompt 'Story:' -value "$docname" story
```

```
readvar -prompt 'Enter &name:' name -prompt 'Enter &age:' age
readvar -prompt 'Enter new margins below:' \
 -prompt 'Lt marg:' -choice '0.5"|1"|1.5"' l \
 -prompt 'Rt marg:' -choice '|same-as-left|0.5"|1"|1.5"' r \
 -prompt 'Top margin:' t \
 -prompt 'Bottom margin:' b
```

redisplay

`redisplay`

The `redisplay` command redraws the Edit pane with the line containing the cursor centered in the window. The redrawing takes effect immediately. This is useful in command scripts to show an intermediate state. By default, changes to windows made inside command scripts do not show up until the script completes.

is a synonym for `redisplay`.

Examples

```
redisplay
red
```

redo

`redo`

The `redo` command reverses the change made by the last `undo` command. A series of consecutive undos may be reversed by the corresponding number of redos. Redo operations do not get added to the undo buffer.

If you choose **Edit ▸ Undo** several times and proceed to perform a new (non undo or redo) operation, the forward items in the undo buffer is discarded in favor of the new operation. In this case there would no longer be any operations to redo.

Note

If you perform an `undo` after a performing a drag and drop move on a large selection, performing a subsequent `redo` may fail.

Example

```
redo
```

remove_file

`remove_file` [`-r` `-f`] *file1* [*file2* ...]

`remove_file` deletes the file(s) specified by *file1*, *file2*, and so on. If a file name specifies a directory, `remove_file` will delete the directory if it is empty. You can only specify a single file at a time from a WebDAV resource. `remove_file` will not remove a group of WebDAV resources or a WebDAV collection (folder).

`remove_file` has the following options:

- `-r` — Specifies that `remove_file` is to recursively delete all files and subdirectories in the specified directories and to delete the named directories themselves.
- `-f` — Specifies that `remove_file` is to delete write-protected files without any warning or prompt.

is a synonym for `remove_file`.

Examples:

```
remove_file myfil
remove_file /usr/draft.xml
remove_file -r -f /usr
remove_file -r -f "$tempfile"
```

If you use a variable name in the command for the file name, place it in double quotes to be sure it's interpreted correctly.

rename_entity

`rename_entity` [*oldname*] [*newname*]

The `rename_entity` alias renames the entity reference *oldname* to *newname*, prompting for input if necessary. This works only for entities declared in the private markup section of the document instance.

re is a synonym for `rename_entity`.

Examples

```
rename_entity mdr ModReq
re arbortext ARBORTEXT
re address cityzip
```

rename_notation

`rename_notation` [*oldname*] [*newname*]

The `rename_notation` alias renames the notation *oldname* to *newname*, prompting for input if necessary. This works only for notations declared in the private markup section of the document instance.

is a synonym for `rename_notation`.

Examples

```
rename_notation CGM cgm
rn Eq equatn
rn TeX tx
```

rename_tag

`rename_tag` *oldtagname newtagname*

The `rename_tag` command renames a tag or entity reference that was defined by the user from *oldtagname* to *newtagname* . The tag name can be made up of any combination of letters, numbers, dashes (-), underscores (_), and periods (.). Entity reference names must start with an ampersand (`&`).

is a synonym for `rename_tag`.

Examples

```
rename_tag comment tech_comment
rt 12-17 12.17.92
rt &address &cityzip
```

repeat

`repeat` [*n*]

The `repeat` command re-executes the previous command typed at the command line or when used in a script, the preceding command, or command list in the script. If a positive integer argument, *n*, is specified, the command will be repeated *n* number of times.

is a synonym for `repeat`.

Examples

```
repeat
rep 5
```

require (Command)

`require` *package* [*filename*]

The `require` command loads the package specified by the variable name *package* from the file specified by *filename*, if it's not already loaded. If *filename* is not specified, the list of directories specified in the `-loadpath` option of the `set` command is searched for a file named `package.acl`, where `.acl` is the standard Arbortext Command Language file name suffix.

If the specified package name is not defined after the file is read, the `require` command will issue an error message and set *main::status* to `1`, indicating failure.

The `require` command has no synonym.

Note

The require command is just like the require function, except that the function allows expressions.

Examples

```
require history
require Tools /ppl/Tools.acl
```

save

`save [-pi | -nopi]`

`save` saves the document that is being edited.

To safeguard against data loss, Arbortext Editor first saves the file to a temporary location; if space is available, the original file is deleted and the temporary file is renamed with the name of the original file.

`save` has the following options:

- `-pi` — Saves Arbortext Editor processing instructions with the document.

  This is the equivalent of setting the `set writepi` on page 935 command to `-all`.
- `-nopi` — Removes all Arbortext Editor processing instructions (except change tracking processing instructions) and may be useful for exporting the document to another system.

  This is the equivalent of setting the `set writepi` command to `-none`.

  Arbortext Editor processing instructions include those representing the `_link` and `_font` tags, user-defined tags, cursor location, detailing state. If you want to save an HTML file with no PIs, you can also use the **File ▸ Save for Browsers** option that is available when editing an HTML 4.0 file.

If the `save`, `write` on page 728 or `save_as` on page 705 commands do not specify whether to write Arbortext Editor processing instructions, they use the value of the `set writepi` command.

Examples

```
save
sav
```

You can also use the `set writepi` on page 935 command to specify whether the `write` on page 728, `save` or `save_as` on page 705 commands save Arbortext Editor processing instructions.

save_all_docs

`save_all_docs` [[`-s`]]

This alias saves the current document and any file entities (including nested entities), regardless of whether the entities have been modified. This alias is especially useful after changing the `set writeentdecls` option for your site.

If `-s` is specified, the message window showing all saved documents is suppressed.

save_as

`save_as` [`-readonly`] { [`-xml`] | `-sgml`} [`-pi` | `-nopi`] [`-encoding` *string*] [*name*]

`save_as` copies the current version of the document or file, including any changes you have made since the last save, to the file name (and path if needed) specified by *name*. The document in the Edit pane will change to the document specified. If *name* exists, a prompt to confirm its replacement will be displayed. To suppress the prompt, prefix the name with an exclamation point (`!`). If *name* is not supplied, the **Save As** dialog box is displayed.

`save_as` has the following options:

- `-readonly` — Saves the document as read-only.
- `-xml` — Saves the document as XML. By default, XML documents are saved as XML documents.
- `-sgml` — Saves the document as SGML. By default, SGML documents are saved as SGML documents.
- `-pi` — Saves Arbortext Editor processing instructions with the document.

  This is the equivalent of setting the `set writepi` on page 935 command to `-all`.
- `-nopi` — Removes all Arbortext Editor processing instructions (except change tracking processing instructions) and may be useful for exporting the document to another system.

  This is the equivalent of setting the `set writepi` command to `-none`.

  Arbortext Editor processing instructions include those representing the `_link` and `_font` tags, user-defined tags, cursor location, detailing state. If you want to save an HTML file with no PIs, you can also use the **File ► Save for Browsers** option that is available when editing an HTML 4.0 file.
- `-encoding` — Saves the file with the encoding specified by *string*; *string* must be one of the following strings:

Supported Encodings

Adobe-Standard-Encoding	Shift_JIS
CNS11643	Big5
ISO-10646-UCS-2	GB2312
ISO-8859-1 to ISO-8859-11	KSC_5601
ISO-8859-13 to ISO-8859-16	EUC-JP
windows-1250 to windows-1258	CEUC
US-ASCII	TEUC
UTF-16	EUC-KR
UTF-8	cp932
cp949	cp936
cp950	

If the command does not specify an encoding, Arbortext Editor sets the encoding using the following rules:

- If the original document is an SGML document and the `-xml` option is specified, the resulting XML file will use UTF-8 encoding. UTF-8 is the default encoding for XML documents.
- If the original document is an XML document and the `-xml` option is specified, the resulting XML file will use the same encoding as the original document.
- If the original document is an XML document and the `-sgml` option is specified, the resulting SGML file will use the encoding used by the operating system.

Note that XML documents that do not contain an encoding declaration in their header are assumed to have the default XML encoding of UTF-8.

Examples

```
save_as
save_as startup.sgm
```

If the `save_as`, `write` on page 728 or `save` on page 704 commands do not specify whether to write Arbortext Editor processing instructions, they use the value of the `set writepi` command.

save_buffers

`save_buffers` [*directory*]

`save_buffers` saves all paste buffers with names to the directory specified by *directory*. If no directory is specified, the current document directory is used.

save_buffers saves all buffers named with the set paste command on page 861 and copy_mark command on page 630 and appends the suffix: .apb. A buffer name may be truncated to conform to operating system limitations.

Buffers are loaded with the load_buffers on page 667 command.

sb is a synonym for save_buffers.

Examples

```
save_buffers
sb bookbfps
```

sh

sh *shell command*

This command passes *shell command*, which is a specified operating system command, to the shell. If the command string contains a semicolon (;) curly braces ({}), a backslash (\) or other special characters, it must be enclosed in quotation marks.

Example

```
sh 'c:\doc\help.com'
```

show aliases (show alias)

show aliases [output=*filename*]

show alias *name* [output=*filename*]

show aliases displays all the currently defined aliases.

show alias *name* displays the definition of the alias named *name*.

If output is specified, this command writes a list of aliases to *filename* where *filename* can be any of the following items:

- The name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file. An exclamation point (!) preceding the file name causes the file to be rewritten, if it exists, without the prompt for confirmation.
- An asterisk (*) specifying the message window. If preceded by a right angle bracket (>), the output is appended to the message window instead of replacing its contents. Additional predefined message windows msgwin2, msgwin3, or msgwin4 use the specifiers output=*2, output=*3, and output=*4, respectively.

- A question mark (?) preceding the output file name. If the file name starts with a question mark, the file name is actually a variable name whose value is set to the output produced by the command. If the second character is a right angle bracket (>), the output is appended to the current value of the variable instead of replacing it.
- A dash (-) specifying standard output. (This is typically the window from which you started Arbortext Editor.)

Examples

```
show aliases
sho aliases output=myaliases
sho aliases output=>allaliases
sho alias rename_entity
sho alias find_id output=myalias.txt
```

show buffers

`show` buffers [output=*filename*]

This command displays a list of all named paste buffers.

If the output option is selected, instead writes a list of named paste buffers to *filename* where *filename* can be any of the following:

- the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.
- an asterisk (*) indicating the message window. If preceded by a right angle bracket (>*), then the output is appended to the message window instead of replacing its contents. Additional predefined message windows msgwin2, msgwin3 , or msgwin4 use the specifier output=*2, output=*3 or output=*4, respectively.

Examples

```
show buffers
sho buffers output=pastebuf.apb
sho buffers output=>pastebuf.apb
```

show characters

`show` characters

This command displays a list of all character entities declared in the DTD with links so that clicking on the entities inserts them in the character entity shortcut menu in the toolbar or in the document if the toolbar is not present.

`show chars` is a synonym for `show characters`.

Example

```
show characters
```

```
show chars
```

show cmdkeys

`show` cmdkeys [output=*filename*]

This command displays a list of command aliases and the associated keyboard shortcut for each command bound to a key. Key assignments to complex commands or functions are not included in the list. The output includes only the default user and system keyboard shortcut assignments.

If the output option is selected, this command instead writes a list of mappings to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of command key assignments to be appended to the end of the file.

Example

```
show cmdkeys
```

show context

`show` context [output=*filename*]

This command lists in a display window the tags and other document structures (such as text or equations) currently valid at the cursor.

Note

If you have applied an alias map to the document, the **Valid inserts at the cursor:** section will include aliases. However, the **Context at the cursor:** string will contain real names only.

If the output option is selected, instead writes the list to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
sho context
sho context output=rules
sho context output=>glblsty
sho context output='>tabify >rules'
```

show emptyelements

`show` emptyelements

This command invokes the Empty Elements dialog box, which lists all elements in the current document that may contain text, but that are empty.

show fullkeymap

`show` fullkeymap [edit | cmd | helpwins | msgwins | textwins | all] [output= *filename*]

This command displays all keymappings (including user-defined and default mappings that have not been remapped) for the Edit pane, command line, help windows, message windows, or all windows. The default is `show fullkeymap edit`.

A list of several windows classes can be made by using a "plus" sign (+) to separate the names. To subtract a window from a list, use a hyphen (-). The option `all` displays the keymappings for the first two windows named above (that is, `edit+cmd`). The `helpwins` option refers to multiple Help windows and is shorthand for: `helpwin1+helpwin2+helpwin3+helpwin4`. The option `msgwins` refers to the predefined message windows and is a synonym for `msgwin1+msgwin2+ msgwin3+msgwin4`. The option `textwins` is a synonym for `helpwins+msgwins`.

If the `output` option is selected, instead writes a list of mappings to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
sho fullkeymap
sho fullkeymap all+text
sho fullkeymap output=/homedir/allkeys
```

show functions

`show` functions [*package*] [output=*filename*]

This command displays a list of all the functions defined in the specified package along with a count of the number of times each function was called in the current session.

- *package* — The name of a loaded package or the fully qualified path name of the file defining the functions. If *package* is not supplied, the current package is used.
- *output* — Writes a list of functions to *filename* where *filename* can be any of the following values:
    - The name of a file. (This could be a complete path name.) A right angle bracket (>) preceding the file name causes the list of aliases to be

appended to the end of the file. An exclamation point (`!`) preceding the file name causes the file to be rewritten, if it exists, without a prompt for confirmation.

- An asterisk (`*`) specifying the message window. If preceded by a right angle bracket (>), the output is appended to the message window instead of replacing its contents. Additional predefined message windows `msgwin2`, `msgwin3`, and `msgwin4` use the specifier `output=*2`, `output=*3` and `output=*4`, respectively.
- A question mark (`?`) preceding an output file name. If the file name starts with a question mark, the file name is actually a variable name whose value is set to the output produced by the command. If the second character is a right angle bracket (>), the output is appended to the current value of the variable instead of replacing it.

Examples:

```
show functions mytools
sho functions c:\acl\functions.acl
```

show ids

`show` ids [output=*filename*]

This command displays the IDs and ID References dialog box that lists all IDs used in `ID` and `IDREF` attribute values and locates any errors in their usage. Attribute values given for `IDREF` must have a matching `ID` attribute of the same value. File (external) entities are included in the search for `ID` and `IDREF` attributes.

> **Note**
>
> If you have applied an alias map to the document, the `show ids` command will display aliases for attribute values that have been assigned aliases.

The menu item **Tools ▸ IDs and ID References** is the same as the command `show ids`. In a file entity Edit pane, selecting **IDs and ID References** or using the `show ids` command displays the same ID list as the parent document. You can't check just the IDs of a file entity unless that entity is edited as a separate document by opening a file entity Edit pane from within the document Edit pane.

If the `-output` option is selected, this command instead writes a list of cross references to *filename*, where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
show ids
show ids output=c:\ch3_lou\crosrefs.txt
```

show keymap

`show` keymap [edit | cmd | helpwins | msgwins | textwins | all] [output=*filename*]

This command displays any user-defined keymappings for the Arbortext Editor window, command line, help windows, message windows, or all, as specified. `show keymap edit` is the default. *all* displays the keymappings for the first two windows named above (that is, *edit*+*cmd*). *helpwins* refers to multiple help windows (if used) and is shorthand for: `helpwin1` + `helpwin2` + `helpwin3` + `helpwin4`. *msgwins* refers to the predefined message windows and is a synonym for `msgwin1` + `msgwin2` + `msgwin3` + `msgwin4`. *textwins* is a synonym for `helpwins` + `msgwins`.

If an *output* value is specified, the command instead writes a list of mappings to *filename* where *filename* can be the name of a file or a complete path and file name. A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
sho keymap
sho keymap edit
sho keymap all
sho keymap output=mymaps.txt
sho keymap output=>smaps.txt
```

show tagnames

`show` tagnames [output=*filename*]

This command displays all valid tag names for the current document type.

Note

If you have applied an alias map to the document, the `show tagnames` command will display aliases for tags that have been assigned aliases.

If the output option is selected, instead writes a list of valid tag names to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
sho tagnames
sho tagnames output=/d1/jkl/publ/booktags.txt
```

```
sho tagnames output=>/d3/syst/publtags.txt
```

show usertags

`show` usertags [output=*filename*]

This command displays user-defined tags.

> **Note**
>
> If you have applied an alias map to the document, the `show usertags` command will display aliases for tags that have been assigned aliases in the **Original Tag** column.

If the output option is selected, instead writes a list of user-defined tags to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file.

Examples

```
sho usertags
sho usertags output=mytags.txt
sho usertags output=>ourtags.txt
```

show variables

`show` variables [output=*filename*]

This command displays a list of all command variables and their values as defined in the current package. For commands entered at the Edit command line, this is always package main.

If the output option is selected, instead writes a list of variables and their values to *filename* where *filename* can be the name of a file (this could be a complete path name). A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file

Examples

```
sho vars
sho vars output=mdkbook.txt
sho vars output=>docsvars.txt
```

source

`source` *filename*

This command reads and executes a list of commands from file *filename*. Reading of the file terminates if a command has an error or if a `return` statement is executed.

If the specified file name does not contain any slashes and is not found in the current directory, the `source` command searches the list of directories specified by the `set loadpath` command on page 848. The extension `.acl` is appended to the file name if it has no extension. For example, if you have a file called `myfuncs.acl` in a directory called `packages` in your home directory, the following will find your file `myfuncs.acl`:

```
append_load_path("~/packages")
source myfuncs
```

If the file *filename* ends with the `.js` extension, then *filename* is passed to the JavaScript interpreter using the `js_source` function on page 397. The JavaScript program is evaluated in the global scope with the *arguments* global object set to null.

If the file *filename* ends with the `.vbs` extension, the file will be run using the `vbscript.dll` script engine using COM.

The abbreviation of `source` is `sou`.

Examples

```
sou cmd.fil
sou /ati/doc/publ/pubdoc.acl
```

spell

`spell` { [`-entityscan` `-noentityscan`] [`-selection` | `-all`] | [`-accept` | `-ignoreall` | `-reject` | `-next` | `-delete`] *word* [*word* …]}

This command checks the entire document, or the selected text, for spelling errors. The default is to check the current selection if there is one, otherwise, the entire document is checked. Misspelled words are displayed in a window.

Examples

```
spell
spe -selection
spe -all
```

`-entityscan (-es)` controls whether or not the `spell` command will include the contents of entities. The counterpart, `-noentityscan (-noes)` will prevent the search from including the contents of entities. For example, the command `spell -es` will check the spelling of the document or selection, including all referenced entities. Setting this option overrides the set entityscan on page 792 command.

The `-accept` option lets you add a new spelling to the “accepted” word list, saved in the `aptspell.xml` file. The `-reject` option lets you add a misspelling to the stop word list, also saved in the `aptspell.xml` file. Note

that these options add words to the `aptspell.xml` file based on the current authoring language. You can set a language on individual tags in a document, so that language can vary based on the position of the cursor.

The `-delete` option removes words from the `aptspell.xml` file.

The `-ignoreall` option will cause Arbortext Editor to ignore the word for the current session, but not save it in the `aptspell.xml` file.

The `-next` option finds the next occurrence of a misspelling when interactive spell checking is enabled.

Arbortext Editor stores the `aptspell.xml` file in the application data directory. That directory is typically `C:\Documents and Settings\`*`username`*`\Application Data\PTC\Arbortext\Editor`.

Elements configured by the `tag_display` command on page 719 to not be displayed can be recognized and operated on by the `spell` command by setting set hiddentagscan on page 829 to `on`.

Examples

```
spell -accept Arbortext
spe -reject AM PM ATI
spe -delete a.m.
```

split (Command)

`split`

This command divides the current element at the cursor into two elements. It searches backward for the first unmatched begin tag and inserts its end tag and another begin tag of the same type.

Example

```
split
spl
```

substitute

`substitute [ -b | -f] [ -c | -noc] [ -a] [ -e | -noe] [ -wrapscan | -nowrapscan] [ -q | -noq] [ -markup]` /*oldtext*/*newtext*/

This command searches for *oldtext* and replaces it with *newtext*.

Note

If you have applied an alias map to the document, the *oldtext* string can include aliases and real names. However, the *newtext* string cannot contain aliases.

- `-a` replaces all occurrences of the string *oldtext* with the string *newtext*. If `-wrapscan` is specified, the command replaces all occurrences in the document. If `-nowrapscan` is specified, the command replaces all occurrences from the current cursor position to the end of the document.
- `-b` searches backward.
- `-f` (the default) searches forward
- `-c` performs a case-sensitive search.
- `-noc` (the default) finds a string regardless of how it is capitalized. The `set case` command on page 761 sets this parameter for all subsequent searches.
- `-e` specifies that the search string is a regular expression. In regular expressions, some punctuation characters (such as `*`, `+`, `?`, `\`, and `.`) perform special actions. For example, `find -e /p..t/` (where the `.` is a placeholder representing any character) would match “pret”, “port”, and “part”, as well as the string “p..t”.
- `-noe` (the default) specifies the search string is not a regular expression. The `set expressions` on page 794 command sets this parameter for all subsequent searches.
- `-wrapscan` (`-ws`) (the default) causes the search to wrap to the beginning of the file when the end is reached.
- `-nowrapscan` (`-nows`) prevents the search from wrapping. The `set wrapscan` on page 927 command sets this parameter for all subsequent searches.
- `-q` (for “quiet”) suppresses the “string not found” message generated when a search fails. This is useful when putting the `find` command in command files or aliases.
- `-noq` (the default) displays a message when Arbortext Editor cannot find a string that matches the search string.
- `-markup` (`-m`) indicates that the *oldtext* and *newtext* strings contain one or more tags or entities. Tags are indicated by surrounding angle brackets; an end tag contains a slash. Entities start with an ampersand (`&`) and end in a semicolon (`;`).

> **Note**
>
> Substituting tag names requires a balanced tag specification. That means if you specify a *newtext* entry of `<para>Beginning of a paragraph`, it's not valid because it's not balanced by a `</para>` end tag. You would need to specify `<para>paragraph text</para>` to be valid.

When performing a case-insensitive substitution (`sub -noc /Never/rarely/`, for example), the replacement is capitalized according to how it was specified, not according to what it replaces (in this example, `Never` and `never` would both be replaced with `rarely`).

You must supply either a matching close delimiter or place the string at the end of a line. For instance, `sub -c /therefore/Therefore` if it's the end of a line and `sub -c /therefore/Therefore/` are both valid.

Examples

```
substitute /the/THE
substitute -b /here/there
s -f /here/there
s -c /here/there/
s -a /and/AND/
s -a -m /<_newline>//
s -e /Eg...t Wil..n/Myrna Jones/
s '(John)(Smith)'\2,\1' -e
```

switch

`switch` (*expression*) { case *constant1*: *statements* break case *constant2*: *statements* break … default: *statements* break}

This command selects from a group of statements based on the value of the expression. The form of the statement is similar to that of the switch statement in the C programming language.

When the `switch` statement is executed, *expression* is evaluated and then control is transferred to the statement following the `case` whose constant value matches the expression. If the expression does not match a `case` value, control is transferred to the statement following the `default` prefix. If there is no `default` prefix, then control resumes following the end of the `switch`.

`Case` is a clause that delimits a set of alternative statements within a switch statement. The `case` and `default` prefixes do not alter flow of control. The break statement may be used to transfer control out of the `switch` statement.

The `case` constant may be a string value delimited by single or double quotes, or a regular expression delimited by slashes. For example, this:

```
switch (x) {
```

```
case 1:
 message "x is 1"
 break
case "one":
 message "x is 'one'"
 break
case /this|that/:
 message "x contains 'this' or 'that'"
 break
}
```

is equivalent to this:

```
if (x == 1) {
 message "x is 1"
} else if (x == "one") {
 message "x is 'one'"
} else if (match(x, "this|that")) {
 message "x contains 'this' or 'that'"
}
```

Since the `match` function is used to evaluate a `case` prefix that specifies a regular expression, the statements following the case may use the match functions `match_result`, `match_start`, and `match_length` to extract parts of the matched expression.

If the `case` constants are all integer values or string constants of a single character, a jump table may be used to transfer control to the matched value. This executes considerably faster than a series of `if,else if` clauses. If the `case` values are all string values, a binary search is used (for more than three cases). A series of `if,else if` statements are compiled if the `case` values are regular expressions or are a mixture of string and integer value.

The body of the `switch` statement must be enclosed in braces. The `case` statements do not need to be enclosed in braces.

Note

The `case` constant label allows for variable substitution. For example:

```
readIncompete=-2
readError=-1
readEOF=0
switch (read(ch, buf, 512)) {
case $readIncomplete:
 ...
case $readError:
 ...
case $readEOF:
 ...
default:
 ...
}
```

tag_display (Command)

`tag_display [-caret|-mouse] [-global|-local] [-default|-full|-partial|-none|-hide|-icon|-toggle] [` *tagname1* [tagname2 …]]

This command tailors the Edit pane display of the tags specified.

Note

If you have applied an alias map to the document, *tagname* can be either an alias or a real name.

If no tag name is given, `-caret` (the default) indicates the first tag to the left of the cursor. `-mouse` indicates the tag under the mouse arrow, or, if the mouse arrow is not on a tag, the first tag to the left of the arrow.

To change the display of every occurrence of a tag, use the `-global` option (the default). One or more tag names may be specified, in which case `-caret` and `-mouse` have no effect.

To change the display of one particular instance of a tag, use the `-local` option. (In this case, the tag must be indicated by the cursor or mouse arrow position; any tag name that does not match results in an error message.)

Note

Local settings have precedence over global ones. A global setting does not affect a tag with a local setting. Use the command `tag_display -local -default` to remove local settings.

The remaining options change the display as follows:

- `-full` — Displays the specified tags in full in all tag display modes.
- `-partial` — Displays the specified tags as the following small icon in all tag display modes: ⋈.

  Singleton tags display as a diamond icon in partial display.
- `-none` — Indicates that the specified tags are not displayed in any tag display mode.
- `-hide` — Hides the specified tags and the contents of these tags in any tag display mode.
- `-icon` — Displays the tags and their contents as the following icon in all tag display modes:
- `-toggle` — (the default) switches between settings as follows:
  - If the global tag display setting for a tag pair is `-icon`, it switches between the thin icon and the previous setting; otherwise, switches between collapsing and exposing detail.
  - For singleton tags, it switches between the thin icon and the previous setting.
- `-default` — Depends on the changes you are making:
  - If you are making global changes to tag display, this option sets the tag display to the default for the tag as set by Arbortext Editor.
  - If you are making local changes to the display of a particular tag, this option sets tag display to any global tag display setting you may have set; if there is none, it sets tag display to the default as set by Arbortext Editor.

The tag display mode is set with the `set tagdisplay` on page 907command.

Elements configured by the `tag_display` command to not be displayed can be recognized and operated on by the `spell` on page 714 and `find` on page 646 commands by setting set hiddentagscan on page 829 to `on`.

is the abbreviation for `tag_display`.

Examples

```
tag_display -icon indexterm
td -default indexterm
td -mouse
```

```
td -icon _ignore
td -local -toggle
td -local -none
td -icon indexterm footnote _comment
td -toggle indexterm
```

time (Command)

`time` [*command* | *command-list*]

The `time` command executes the specified command(s) and reports how much real (clock) user and system CPU time has elapsed. (System time is always zero on Arbortext Editor for Windows.) If no argument is given, then `time` reports the times since Arbortext Editor started.

toggle

`toggle` [*optionname*]

This command changes the value of the Arbortext Editor boolean option specified by *optionname*. The specified *optionname* must be a boolean option, having only two values (for example, on/off, 1/0, yes/no, etc.). This command evaluates the current setting and sets the option to its opposite setting.

Note

This command will not work with options that have two settings that are specific string names. For example, this command will not change `diffencltype`, since the available values are explicitly `pi` and `tag`, not one (1) and zero (0).

Example:

```
toggle tabletags
toggle addrequiredtags
```

translate

`translate` { uc | lc | mc | pc}

This command converts the case of a highlighted region (or the character before the cursor if no region is selected) in the following manner:

- `translate uc` (uppercase) converts every letter in the region to an uppercase or capital letter.
- `translate lc` (lowercase) converts every letter in the region to a lowercase letter.
- `translate mc` (mixed case) capitalizes the first word of every sentence by converting the first character that follows a period, question mark, or exclamation point to a capital letter if it is a letter.
- `translate pc` (proper case) capitalizes the first letter of every word.

Examples

```
translate uc
tra lc
tra mc
tra pc
```

unalias

`unalias` alias

Removes the specified alias from the current list of aliases.

Examples

```
unalias pscr
```

undeclare_entity

`undeclare_entity` [*name*]

This alias removes the user-defined entity reference *name* from the private markup section of the document instance, thus eliminating the entity. If the entity name is not supplied, you are prompted for it.

To use `undeclare_entity`, you must first remove all references to the entity in the document. If the entity is referenced in the document's paste buffer, you are prompted to clear the paste buffer.

is a synonym for `undeclare_entity`.

Examples

```
undeclare_entity
ude atext
```

undeclare_notation

`undeclare_notation` [*notationname*]

This alias removes the user-defined notation *notationname* from the SGML Declaration in the private markup section of the document instance, thus eliminating the notation. If the notation name is not supplied, you are prompted for it.

is a synonym for `undeclare_notation`.

Examples

```
undeclare_notation
udn
```

undefine_keymap

`undefine_keymap` *name*

This command deletes the keymap named *name*, which was created by `define_keymap` or `copy_keymap`. The keymap must not be currently attached to a window. The command will fail if it is. The function `keymap_exists` may be used to determine if the keymap is in use.

is a synonym for `undefine_keymap`.

Example

```
undefine_keymap panel_map
```

undefine_tag

`undefine_tag` *tagname*

This command undefines the user tag *tagname* (or entity reference) defined previously by the user.

is a synonym for `undefine_tag`.

Examples

```
undefine_tag indxterm
udft cologo
```

undo

`undo` [boundary | current | clear | mark | unmark | suspend | resume]

With no arguments, this command undoes the most recent change. A `redo` brings it back.

The options are only useful within ACL scripts. The `boundary` option causes Arbortext Editor to insert a boundary in the undo history so that a subsequent undo will restore changes up to the current change. Normally, when executing a function or alias mapped to a key or menu item, Arbortext Editor considers all

changes made by the script as a single undoable event. The function or alias can use `undo boundary` to cause the changes to be considered as separately undoable operations.

If `current` is specified, then this command undoes all changes made by the currently executing function or alias and discards the undo history. This can be used to cancel the effect of a command script, for example, if the script prompts for confirmation and the user cancels the request.

If `clear` is specified, the undo buffer is cleared. You will not be able to undo any prior operations. Subsequent operations will begin to re-fill the undo buffer.

The `mark` option inserts a temporary boundary in the undo history that can be removed with the `unmark` option. This is useful if a script makes changes to the document and then raises a modal dialog to make additional changes. Normally, when a dialog is raised Arbortext Editor inserts a boundary in the undo history. This means the changes made before the dialog was raised would require a separate undo operation to undo. The script can use `undo mark` and `undo unmark` to cause the two sets of changes to be combined into a single undo event. See the example below.

If `suspend` is specified, then any changes made after the command will not be added to the undo history and cannot be undone. Use `undo resume` to re-enable the undo history. You should use the `undo suspend` command with care, since it is possible to disable undo history for all further edits to the document if `undo resume` is not called.

Undo operations are not added to the undo buffer.

Examples

```
und
undo boundary
undo current
# undo mark/unmark/current example:
if (insert_tag(name)) {
 # Tag insertion succeeded, set a temporary boundary and
 # raise the modify attributes dialog as a modal dialog
 undo mark;
 modify_tag -local -modal;
 if (main::status != 0) {
#undo current;  # Undo cancelled so back out insert_tag change
} else {
 # Dialog succeeded, so remove the temporary undo boundary so
 # both operations will become a single undoable event.
 undo unmark;
}
}
```

unmap

`unmap` [window] keyname

This command removes the current user-defined mapping from the specified key. The `window` argument identifies which window classes are to be affected by unmapping `keyname`. The keymap attached to the current window is the default.

This command cannot be used to unmap Arbortext Editor's default key mappings. For example, it is impossible to use the unmap command to make the **F1** key do nothing.

The value for `window`, which is a window class name or user-defined keymap, can be any of the following:

- `edit` designates the Edit pane
- `cmd` designates the command line
- `helpwin` (also called `helpwin1`), `helpwin2`, `helpwin3`, `helpwin4`, `msg` (also called `msgwin1`), `msgwin2`, `msgwin3`, and `msgwin4` designate multiple Help windows. (Use `msgwin` for Arbortext Editor messages and output from the `show`, `eval`, and similar commands.)
- `window` name as returned by the `window_name` function
- `@` name designates the name of a user-defined keymap.

If `window` is a class name, the mapping affects all windows using the specified class keymap, otherwise `unmap` changes the mapping in the keymap defined by a previous `define_keymap` or `copy_keymap` command.

A list of several window classes can be made by using a “plus” sign (+) to separate the names. To subtract a window class from a list, use a hyphen (−). The option `all` is a synonym for `edit+cmd`. Note that the `helpwins` and `msgwins` are not included in the `all` option. To map a key for all possible window classes, use: `map all+textwins`. The `textwins` option is a synonym for `helpwins+msgwins`. The `helpwins` option is shorthand for `helpwin1+helpwin2+helpwin3+helpwin4`.

Examples

```
unmap f1
unm cmd backspace
unm helpwin2 Tab
unm edit+cmd f1
unm all+textwins-cmd shift-UpArrow
```

unsetvar

`unsetvar` *variable1* [*variable2* …]

This command removes any setting for the variable specified. If an array name is specified, all elements are discarded. Individual array elements may be removed using the `delete` built-in function. Note the absence of the leading dollar sign.

Examples

```
unsetvar docname
unsetvar docname user
```

version

`version`

Shows the version panel for the version of the software that is being run.

Examples

```
version
ver
```

wait

`wait` *n*

This command delays *n* seconds before executing the next command.

Example

```
wait 3
```

while

`while` (*condition*) {*cmds*}

This command repeats the commands listed as long as the given condition is true.

cmds is a list of one or more commands. The list must be enclosed in curly braces. *condition* is a logical expression describing the case in which these commands are to be executed.

Note

It is essential that the commands that are executed are ones that eventually change the condition so that it is no longer true. Otherwise, Arbortext Editor will stop responding, being in an infinite loop.

Examples

```
find -t '<emphasis>Editor</emphasis>'
while ($status == 0) {
 delete_mark; insert_entity ed; find
}
```

window

`window` { *pct* | */string/* | down | up | *line*,*col*} [`-c` | `-noc`] [`-e` | `-noe`] [`-t` | `-not`] [`-wrapscan` | `-nowrapscan`] [`-q` | `-noq`] `-nocaret` `-wheelup` `-wheeldown`

This command positions the Edit pane so the line specified by the argument is the first line on the screen. The `-pct` argument specifies a number between 0 and 100 and represents the percentage in the document that should be scrolled to the top of the screen. If 0, the beginning of the document is displayed; if 100, the end is scrolled into view.

If `/-string/` is specified, the line containing the next occurrence of the specified string (or pattern if `-e` is also given) is scrolled to the top of the screen. Any of the standard string delimiters may be used in place of " /".

If `-down` is specified, the document is scrolled so the bottom line on the screen is moved to the top, corresponding to the **PAGE DOWN** key.

If `-up` is specified, the document is scrolled so the top line on the screen is moved to the bottom, corresponding to the **PAGE UP** key.

If `-wheeldown` is specified, the document is scrolled down the number of lines that correspond to the current setting for the scroll wheel on the pointer device, usually three lines.

If `-wheelup` is specified, the document is scrolled up the number of lines that correspond to the current setting for the scroll wheel on the pointer device, usually three lines.

If `-line,col` is specified, the line containing the character at the specified location is scrolled to the first line on the screen. Allowable values for `-line` and `-col` are the same as for the `caret` on page 621 command. However, only relative line numbers may be specified.

The `-nocaret` modifier specifies that the cursor position should not be changed even if it is off the screen after the document is scrolled to the new top line.

The other modifiers here work like the modifiers for the `find` on page 646command.

Examples

```
window 50
win 100
win up
win /<section>/ -t
win +2,0
win bottom+1,0
win -2,0
win bottom,0
win (bottom-top)/2,0
```

write (Command)

`write` [`-ct` [original | changesapplied | changeshighlighted]] [`-sgml` | `-untagged` | `-xml`] [`-encoding` *string*] [`-paste` | `-buffer` *buffername* | `-all`] [`-ok`] [`-header` | `-noheader`] [`-pi` | `-nopi`] [`-eoc` | `-noeoc`] [`-nonasciichar` [entref | numref | char]] [`-public` *pubident*] [`-sysid` *sysident*] [`-nobreakattag`] [`-flatten` [file | text | both]] *filename*

The `write` command saves a version of the document, or a portion of the document, to the file specified by *filename*.

Note

The `write` command does not test for available space. If you need to test for available space, use the `save` on page 704 command.

filename can be any of the following:

- The name of a file, including path if needed.

  A right angle bracket (>) preceding the file name causes the list of aliases to be appended to the end of the file. An exclamation point (`!`) preceding the file name causes the file to be rewritten, if it exists, without the prompt for confirmation.
- An asterisk (`*`) indicating the message window. If preceded by a right angle bracket (`>*`), the output is appended to the message window instead of replacing its contents.

The `write` command has the following options:

- `-ct` — Specifies how the document will write changes that have been made with change tracking turned on:
  - *original* — Writes the document as if all pending changes were rejected.
  - *changesapplied* — Writes the document as if all pending changes were accepted.
  - *changeshighlighted* — Writes the document with all pending changes displayed.

  The `write` command defaults to the `set writechangetracking` on page 928 command setting. If `set writechangetracking` is not set, the `write` command writes out the document with all pending changes highlighted (that is, using *changeshighlighted*).

The save on page 704 and save_as on page 705 commands will always write the change tracking information.

- `-sgml` — Writes a version of the document conforming to the SGML standard, ISO 8879. Unless the `-noheader` option is specified, the standard DOCTYPE header listing any private ENTITY declarations will appear at the top of this version. By default, SGML documents are written as SGML documents.
- `-untagged` — Writes a text-only version of the document which contains no tags. No entity expansion or substitution is attempted. `write -untagged` does not write equations.
- `-xml` — Writes the file as an XML document. By default, XML documents are written as XML documents.
- `-encoding` — Specifies with *string* the encoding of the file being written. The setting of this option overrides the encoding declaration in an XML file.

  string must be one of the following encoding strings:

  ### Supported Encodings

Adobe-Standard-Encoding	Shift_JIS
CNS11643	Big5
ISO-10646-UCS-2	GB2312
ISO-8859-1 to ISO-8859-11	KSC_5601
ISO-8859-13 to ISO-8859-16	EUC-JP
windows-1250 to windows-1258	CEUC
US-ASCII	TEUC
UTF-16	EUC-KR
UTF-8	cp932
cp949	cp936
cp950	

  If the command does not specify an encoding, Arbortext Editor sets the encoding using the following rules:

  - If the original document is an SGML document and the `-xml` option is specified, the resulting XML file will use the original encoding if the SGML document has a byte-order mark (an ISO-10646-UCS-2 file) or a special encoding was set using `edit -encoding`. If there was no special encoding or it isn't an ISO-10646-UCS-2 file, then the resulting XML file will use UTF-8 encoding. UTF-8 is the default encoding for XML documents.

- If the original document is an SGML document and either no option is specified or the `-sgml` option is specified, the resulting SGML file will use the same encoding as the original document.
- If the original document is an XML document and the `-sgml` option is specified, the resulting SGML file will use the encoding used by the operating system.
- If the original document is an XML document and either no option is specified or the `-xml` option is specified, the resulting XML file will use the same encoding as the original document.

XML documents that do not contain an encoding declaration in their header are assumed to have the default XML encoding of UTF-8.

- `-paste` — Writes the contents of the current paste buffer to the specified file.
- `-buffer` — Writes the contents of the named paste buffer to the specified file.
- `-all` — Writes the entire document to the new file name. This is the default option.

Note

If the paste buffer includes a marked section parameter entity, it is written as an undefined marked section parameter. Use the `modify_ms_parameters` on page 683 (or its abbreviated version `mmsp`) command on the command line to define it.

- `-ok` — Writes the file without prompting you if a file with the same name already exists. The existing file will be overwritten. Alternatively, the file name may be preceded with an exclamation point (!) to suppress the prompt.
- `-header` — Specifies that the DOCTYPE header and any private ENTITY declarations that normally appear at the top of a standard DOCTYPE header are maintained.
- `-noheader` — Removes the DOCTYPE header and any private ENTITY declarations that normally appear at the top of a standard DOCTYPE header.
- `-pi` — Writes processing instructions specific to Arbortext Editor.

  This is the equivalent of setting the `set writepi` on page 935 command to `-all`.

  If the `save_as` on page 705, `write` or `save` on page 704 commands do not specify whether to write Arbortext Editor processing instructions, they use the value of the `set writepi` command.

- `-nopi` — Removes all processing instructions (except change tracking processing instructions) specific to Arbortext Editor.

  This is the equivalent of setting the `set writepi` command to `-none`.

  Arbortext Editor processing instructions include those representing the `_link` and `_font` tags, user-defined tags, cursor location, detailing state. If you want to save an HTML file with no PIs, you can also use the **File ▸ Save for Browsers** option that is available when editing an HTML 4.0 file.

  With the `-paste` option, the `write` command also sends fragment processing instruction output indicating the context of the fragment in the paste buffer. When Arbortext Editor opens such a file, a fragment tag pair carrying this context information surrounds the contents of the file.
- `-eoc` — Turns on entity conversion, letting you specify how you want Arbortext Editor to write out character entities. This setting overrides the value of the `set entityoutputconvert` command on page 791.
- `-noeoc` — Turns off entity conversion, and, if `set entityinputconvert` is `off`, Arbortext Editor writes out character entities as they were read in. This option overrides the value of the `set entityoutputconvert` command on page 791.
- `-nonasciichar` — Specifies how Arbortext Editor writes out non-ASCII characters. `-nonasciichar` has the following arguments:
  - `entref` — Writes out non-ASCII characters as character entity references. If Arbortext Editor cannot find matching character entity references, it writes out the non-ASCII characters as numeric character references. This is the default setting.
  - `char` — Writes out non-ASCII characters as characters in the target encoding. If Arbortext Editor cannot find the characters in the target encoding, it writes out the non-ASCII characters as numeric character references.
  - `numref` — Writes out non-ASCII characters as numeric character references.

  The setting of the `-nonasciichar` option overrides the value of the `set writenonasciichar` command on page 933.

  A synonym for `-nonasciichar` is `-nac`.
- `-public` — Writes the alternate public identifier *pubident* on the `DOCTYPE` declaration instead of the original value (if any). *pubident* is not checked for validity as a minimum literal. If *pubident* is `<none>`, then the `PUBLIC` identifier will be omitted.

- `-sysid` — Writes the alternate system identifier *sysident* on the `DOCTYPE` declaration instead of the original value if any. For XML files, *sysident* should be a valid URL (as created by the `filename_to_url` on page 358 function). If *sysident* is `<none>`, the `SYSTEM` identifier will be omitted. Using this option overrides the setting of the `set writeabsolutesysid` command on page 927.
- `-nobreakattag` — Specifies that breaks not be inserted at element boundaries or before the tag close character (applies to XML documents only). Such breaks may not be properly ignored by web browsers. Avoiding breaks at element boundaries tends to produce many breaks within elements. This may produce a file that is difficult to read when viewed with a text editor. Using this option overrides the setting of the `set writenobreakattag` command on page 933.
- `-flatten` — Specifies that Arbortext Editor expand entity references on write, replacing references with the text of the entity type as specified by one of the following arguments:
  - `file` — Expands all file entities recursively. Declarations for all file entities are omitted from the internal subset.
  - `text` — Expands all text entities recursively. Declarations for all text entities are omitted from the internal subset.
  - `both` — Expands both file and text entities. This is a shorthand combination of `-flatten file -flatten text`.

Examples

```
write -untag -paste cherdoc.txt
write -sgml proposal.xmm
write -buffer bufC oldver.sgm
```

xmsgfmt

`xmsgfmt` [`-o` *filename*.amo] [*file1*.xlf] [*file2*.xlf ...]

`xmsgfmt` reads XLIFF `.xlf` files and creates a `.amo` file that can be used by Arbortext Editor for all error messages it displays.

- `-o` *filename*`.amo` — The `.amo` file to be created. If not given, the output is written to *`file1`*`.amo`
- *`file1`*`.xlf` — One or more XLIFF files to be compiled into an `.amo` file.

6

set Command Options

set

`set [-all] [-session] [-preference] [-local] [option=value]`

Sets the desired *option* to the desired *value*.

The `-all`, `-session`, `-preference` and `-local` options control the `set` option scope on page 136. These options are mutually exclusive. Use only one at a time.

- `-all` — Sets the session scope of the `set` option and the local scope (if any). All open windows redraw to reflect the update.
- `-session` — Sets the session scope for the `set` option, leaving the local scope unchanged. This option is useful for changing a setting so it affects new windows, but does not change the appearance of currently open windows or the value used when determining what gets written to your preferences file.
- `-preference` — Sets the session scope for the set option, leaving the local scope unchanged, and sets the value used when determining what gets written to your preferences file. This option is useful for preparing to call the `write_preferences` on page 608 function when updating just an option's preference but not other session-specific values that may have changed.
- `-local` — Sets the local scope for the `set` option, leaving the session scope unchanged. This option is useful for changing the appearance of the current view or document.

Many `set` options are available in the user interface using the **Advanced Preferences** dialog box. (Choose **Tools ► Preferences** and then choose the **Advanced** button.)

The complete list of `set` command options are available from the online help table of contents by choosing **Arbortext Command Language (ACL) ► Commands ► Commands ► set**. The command options are also listed alphabetically in the online help index.

set acceptcmdextension

`set` acceptcmdextension= { on | off}

This option determines whether the `source` and `require` commands will add the `.cmd` extension to the file name if no extension was specified. If `on` , the `.cmd` extension will be tried after `.acl` if the file is not found. If `off` (the default), only `.acl` is supplied.

Note

The `.cmd` extension is deprecated and should not be used for new applications. When searching directories beneath *$aptpath*, Arbortext Editor never adds the `.cmd` extension regardless of the `acceptcmdextension` setting.

The option does not affect files specified explicitly with the `.cmd` extension. For example:

```
source myfuncs.cmd
```

set accessibility

`set` accessibility= { on | off}

If this option is set to `on`, Arbortext Editor operates in a manner compatible with screen readers such as JAWS. This option allows screen readers to read menus more easily, and the edit cursor displays as a line one pixel thick.

You can set this option as a preference from the **Tools ► Preferences** dialog box. Click the **Advanced** button and choose `set accessibility`. Click the option button `on` in the dialog box.

You can also set it in an ACL script loaded from the *`Arbortext-path`*\custom \init directory or from the command line (available if **Full Menus** is checked in the **Tools ► Preferences ► Window** dialog box).

If a screen reader can notify the system that it has been started or stopped, the `set accessibility` option switches between `off` and `on` if the screen reader starts or stops while Arbortext Editor is running.

Note that as of 4.51, JAWS does not have this capability. You can either turn on the `set accessibility` option or make sure that JAWS is running before you start Arbortext Editor.

set addrequiredtags

`set` addrequiredtags= { on | off}

When set to `on`, this command inserts any required elements at the same time a tag is entered (as long as such insertion is not ambiguous). The default setting is `on`.

is a synonym for `addrequiredtags`.

set aliaslocale

`set` aliaslocale= localename

This command specifies a *`localename`* (such as French or German) for a document type's alias map file. The default is the system locale, which is defined by the operating system. Arbortext Editor will try to find the map file specified by the `aliasmap` option in *`doctypename`*`\locale\`*`localename`*, where *`doctypename`* is the directory in which the document type is located. If Arbortext Editor does not find *`doctypename`*`\locale\`*`localname`*, or the map file cannot be found in the *localename* subdirectory, Arbortext Editor will search for the map file in the *`doctypename`* directory.

For example,

```
set aliaslocale=French
set aliasmap=docbookfr
```

where `docbookfr` is an alias map file located in the *`doctypename`*`\locale\French` directory.

Note

This command will not have any effect if the `aliasmap` option specifies a full path for the map file.

set aliasmap

`set` aliasmap= { none | *mapname*}

Use this command to apply or remove an alias map for the current document. *`mapname`* can be a file name or the full path name of an alias map file. If *`mapname`* is a file name, Arbortext Editor appends an `.alias` suffix to it and tries to locate the file in the *`doctypename`*`\locale\`*`localename`* directory, where *`doctypename`* is the directory in which the document type is located. If the file does not exist in this location, Arbortext Editor tries to locate it in the *`doctypename`* directory.

Every time you change the `aliasmap` setting, the Edit Window will be refreshed to reflect the change.

You can add the `set aliasmap` command to the `instance.acl` file in the *`doctypename`* directory to specify a default alias map.

Note

If there are dialog boxes open in the Edit window, Arbortext Editor closes them before executing the `set aliasmap` command.

set allowinvalidmarkup

`set` allowinvalidmarkup= { on | off}

This command controls whether invalid markup is allowed in during the loading of a file. If set to , a file containing invalid markup would lose the invalid markup at load time.

The default is `on`.

set allowsvgexternalresources

`set` allowsvgexternalresources= { on | off}

This command controls whether links to external resources in SVG images are supported. If set to off, Arbortext Editor throws an error when previewing or publishing such SVG file containing external resources.

The default is `off`.

set appconfigfile

`set` appconfigfile=*filename*

This command specifies an alternate location for an `.appcf` PDF configuration file for the PTC APP engine.

The command may have one of three values:

- `" "` (empty string) — the command is ignored
- absolute path to an `.appcf` PDF configuration file

  Local composition: Arbortext Styler will attempt to use the file indicated. If the file does not exist, cannot be read, is not an PTC APP configuration file, or is an PTC APP configuration file but does not support the type of operation being performed, it is ignored.

PE composition: Arbortext Editor or Arbortext Styler will look for an exact match in the `list of known PTC APP configuration files` for the PE server. If there is a match, and the file in question supports the type of operation being performed, the file will be used. Otherwise the option will be ignored.

- simple filename, i.e. *`filename`*`.appcf`

  Both local and PE composition will scan the `list of known PTC APP configuration files` to find a matching filename that supports the kind of operation being performed. If there is a match., that PTC APP configuration file will be used. If there's no match, this option will be ignored.

The `list of known PTC APP configuration files` is built by searching directories on the PE server (for PE composition) or directories on the local machine (for local composition) for APP configuration files (`.appcf`). Files found in a directory are added to the list in alphabetical order.

The directory search order is:

1. (Local composition only) The directory containing the document instance
2. Each custom doctype directory for the document being composed

   If the document's doctype is `D`, and the list of custom directories contains `c1`, `c2`, and `c3`, the directories `c1/doctypes/D`, `c2/doctypes/D`, and `c3/doctypes/D` will be searched.

   The custom directory search order is determined by the order they are declared in `APTAPPLICATION` and `APTCUSTOM`.
3. The document's doctype directory in *`Arbortext-path`*
4. `app` subdirectories of each custom directory (order of custom directories is determined as specified in (2) above
5. The `app` subdirectory in *`Arbortext-path`*
6. (Local composition only) Any PTC APP configuration files explicitly browsed during the current Arbortext Editor or Arbortext Styler session

Note

If PE composition is enabled, the custom doctype and `app` directories, and the *`Arbortext-path`* doctype and `app` directories, are on the PE server. There is no way to choose an PTC APP configuration file on the local machine and have it transmitted to the PE server.

Example:

```
set appconfigfile=memo.appcf
```

set appsnapshot

`set` appsnapshot= { yes | no}

When set to `yes`, this command collects the transformed XML document that was submitted to PTC APP for printing or publishing to PDF into a zip archive for troubleshooting. The collection also includes the graphics it references and other supporting files. When you set this command to `yes`, the publishing process creates a zip for each subsequent publishing operation that uses PTC APP. The default setting is `no`.

The PTC APP snapshot file contains the following:

- `manifest.xml` — An XML document that contains information about the publishing parameters in effect when the publishing request was made.
- `inputDoc.xml` — The document sent to PTC APP that has been modified for troubleshooting. It will be flattened and its graphic references will point to the `doc-graphics` directory.
- `doc-graphics` — A directory containing a copy of the graphics referenced by `inputDoc.xml`.
- `appsnap.log` — A log of the events from generating the PTC APP snapshot. If the snapshot is not complete, the `appsnap.log` will contain information about it.
- `dynamic.dtd` or `dynamic.xsd` — An XML document type definition (DTD) file or an XML schema (XSD) file may be necessary to read `inputDoc.xml`. The snapshot may contain one of these files if the snapshot also contains an `inputDoc.xml`.

> **Note**
>
> DITA documents will use a dynamic DTD rather than a schema.

The location where the PTC APP snapshot zip file is saved depends on how the publishing request is made.

- If you are sending publishing requests to Arbortext Publishing Engine, the snapshot will be generated and the transaction will be archived on the Arbortext PE server. If `set debugcomposition` on page 770 is set to `on`, then the snapshot file will be included with the intermediate files returned to the Arbortext Editor client.

See Using Arbortext Publishing Engine for Publishing Documents for information.

- If you are publishing locally using Arbortext Editor with Arbortext Styler, the PTC APP snapshot is placed in the Cache Directory (.aptcache). If you use the Editor menu choices for publishing PDF or printing, a dialog box will confirm the action and notify you where the snapshot has been saved. It will also open the target directory for you if you click **Browse to directory**. See `set appsnapshotprompt` on page 750 for more information.

  If `set debugcomposition` on page 770 is set to `on`, then the snapshot will also be included in the `compose.zip` file produced from **Tools ► Save Application**. See **Save Application dialog box** for more information.

You can run `set appsnapshot` from an ACL script loaded from the *`Arbortext-path`*\custom\init directory or from the command line in Editor (available if **Full Menus** is checked in the **Tools ► Preferences ► Window** dialog box).

Consult Troubleshooting PTC APP Publishing for more information. Using snapshots for troubleshooting should be done with the help of Technical Support.

set appsnapshotprompt

`set` appsnapshotprompt= { yes | no}

When set to `yes` (the default), this command will launch a dialog box confirming the action that an PTC APP snapshot has been generated and giving its location. Refer to `set appsnapshot` on page 749 for more information.

This command applies to the Editor print and publishing to PDF outputs. Set it to `no` if you are publishing using publishing rules or from a script.

Consult Troubleshooting PTC APP Publishing for more information. Using snapshots for troubleshooting should be done with the help of Technical Support.

set asciiattributecolor

`set` asciiattributecolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display attributes in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `blue`.

Examples

```
set asciiattributecolor=blue
set asciiattributecolor=#dd00dd
```

set asciicommentcolor

`set` asciicommentcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display comments in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `violet`.

Examples

```
set asciicommentcolor=blue
set asciicommentcolor=#dd00dd
```

set asciideclarationcolor

`set` asciideclarationcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display declarations and processing instructions in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `green`.

Examples

```
set asciideclarationcolor=blue
set asciideclarationcolor=#dd00dd
```

set asciiendtagcolor

`set` asciiendtagcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display end tags in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `red`.

Examples

```
set asciiendtagcolor=blue
set asciiendtagcolor=#dd00dd
```

set asciientitycolor

`set` asciientitycolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display entity references in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `olive`.

Examples

```
set asciientitycolor=blue
set asciientitycolor=#dd00dd
```

set asciiextension

`set` asciiextension= *ext*

This command specifies the extension to use for newly saved ASCII (non-SGML) files. The default value is `txt`. The **Save As** dialog box and `save_as` command add this extension if the specified file does not already have an extension and the option is non-null.

A period should not be included as part of the extension.

Examples

```
set asciiextension=txt
```

set asciistarttagcolor

`set` asciistarttagcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display start tags in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `red`.

Examples

```
set asciistarttagcolor=blue
set asciistarttagcolor=#dd00dd
```

set asciisyntaxcolor

`set` asciisyntaxcolor= { on | off}

This command determines whether ASCII documents with markup are displayed with colors to distinguish different markup constructs. The default is `on`.

set asciitextcolor

`set` asciitextcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display text in ASCII documents with markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `black`.

Examples

```
set asciitextcolor=blue
set asciitextcolor=#dd00dd
```

set asciixslresultattributecolor

`set` asciixslresultattributecolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display result tree attributes in ASCII documents with XSL markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `teal`.

Examples

```
set asciixslresultattributecolor=blue
set asciixslresultattributecolor=#dd00dd
```

set asciixslresultendtagcolor

`set` asciixslresultendtagcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display result tree end tags in ASCII documents with XSL markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `maroon`.

Examples

```
set asciixslresultendtagcolor=blue
set asciixslresultendtagcolor=#dd00dd
```

set asciixslresultstarttagcolor

`set` asciixslresultstarttagcolor= { *colorname* | *#rgbspec*}

This option determines the color value used to display result tree start tags in ASCII documents with XSL markup when `asciisyntaxcolor` is set to `on`. The value is either a named color or an RGB specification preceded by `#`. The default is `maroon`.

Examples

```
set asciixslresultstarttagcolor=blue
set asciixslresultstarttagcolor=#dd00dd
```

set autocorrect

`set` autocorrect= { on | off}

When the `autocorrect` option is set to `on`, each word entered or changed when editing a document is looked up in the `autocorrect.xlf` auto correct map file. If that word is found in the map file, the word is automatically replaced by the replacement word specified in the auto correct map file. The default setting is `on`.

Arbortext Editor can support multiple language-based versions of `autocorrect.xlf`. For more information, see autocorrect.xlf.

set autosave

`set` autosave= { *n* | off}

When the `autosave` option is set to a positive number, Arbortext Editor saves any changes to the document every `n` seconds if there are unsaved changes. `600`, or 10 minutes, is the default. The maximum setting is 7200, or 2 hours.

When Arbortext Editor performs an autosave for a document, it doesn't write to the original file. Instead, Arbortext Editor saves the document to a temporary file (based on the document name) with the extension `.asv`. If this file exists when the document is reopened, Arbortext Editor will prompt for which version to use. The autosave file, if any, is removed when the document is saved.

Examples

```
set autosave=60
set autosave=off
```

set autotaginserts

`set` autotaginserts= { on | off}

When the `autotaginserts` option is set to `on`, it obeys the `.dcf` file setting for automatic tag insertion.

When set to `off`, this command disables automatic tag insertion.

set backgroundcoloraqua

`set` backgroundcoloraqua= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **aqua**. The value is either a named color or an RGB specification preceded by `#`.

set backgroundcolorblack

`set` backgroundcolorblack= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **black**. The value is either a named color or an RGB specification preceded by `#`.

set backgroundcolorblue

`set` backgroundcolorblue= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **blue**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorbrown

`set` backgroundcolorbrown= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **brown**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray

`set` backgroundcolorgray= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray1

`set` backgroundcolorgray1= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray1**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray2

`set` backgroundcolorgray2= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray2**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray3

`set` backgroundcolorgray3= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray3**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray4

`set` backgroundcolorgray4= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray4**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgray5

`set` backgroundcolorgray5= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **gray5**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorgreen

`set` backgroundcolorgreen= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **green**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorlime

`set` backgroundcolorlime= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **lime**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolormaroon

`set` backgroundcolormaroon= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **maroon**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolornavy

`set` backgroundcolornavy= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **navy**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorolive

`set` backgroundcolorolive= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **olive**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolororange

`set` backgroundcolororange= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **orange**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorred

`set` backgroundcolorred= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **red**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorteal

`set` backgroundcolorteal= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **teal**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorviolet

`set` backgroundcolorviolet= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **violet**. The value is either a named color or an RGB specification preceded by #.

set backgroundcolorwhite

`set` backgroundcolorwhite= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **white**. The value is either a named color or an RGB specification preceded by `#`.

set backgroundcoloryellow

`set` backgroundcoloryellow= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the background (shading) color named **yellow**. The value is either a named color or an RGB specification preceded by `#`.

set balancedselections

`set` balancedselections= { on | off}

When the `balancedselections` option is set to `on`, it automatically extends the selection so that tags are balanced. For example, if you select either a start or an end tag when this option is set, the region will be expanded to include the entire tag pair. The default is `on`.

When the `balancedselections` option is set to `off` and an unbalanced selection portion of markup is deleted, all selected content and balanced portions of the selection are deleted. However, no tagging is removed that would result in unbalanced markup or out-of-context errors.

set bigjobthreshold

`set` bigjobthreshold= { *n* | off}

This option helps determine if and when the output from a content pipeline is too large to be written to memory, and should be written to disk instead. When the number of document fragments in the document being published exceeds the threshold value *n*, the document is considered large enough to write the content pipeline output to disk rather than to memory. Setting *n* to `0` causes pipeline output to be written to disk for all publishing requests.

This set option has document scope. The default value is `off`.

Some documents are so large that a content pipeline may not be able to write its output to memory. Writing a document into memory uses more memory than reading a document from disk into memory. Arbortext Editor and Arbortext

Publishing Engine normally write output to memory, but can, under specified conditions, write pipeline output for a large document to disk to minimize memory consumption.

If content pipeline output is written to disk, then Arbortext Editor or the Arbortext PE sub-process immediately reads the document from disk into memory and proceeds as if the content pipeline had written the document into memory. After the pipeline output is in memory (regardless of the method used), processing continues with other tasks, such as running the formatter, the HTML Help compiler, or some other action.

Set the level at which the pipeline output is written to disk by specifying *n* for the number of document fragments (tags, text entities, file entities, and XML inclusions) in the document. For example, if you `set bigjobthreshold=25`, any document containing more than 25 document fragments will be written to disk during pipeline processing.

The ACL function `doc_estimate_dfs` on page 317 can be helpful in determining what values to use for the `bigjobthreshold` option.

When publishing a DITA map document, the overall size of the document produced by the content pipeline is determined by the size of the DITA topics that are referenced from the map rather than by the size of the map itself. To help account for this difference, the number of document fragments in the map is multiplied by 80 before being compared to the threshold. This approximation allows a common threshold to be used for DITA topic, DITA map, and non-DITA document publishing. This means the `bigjobthreshold` value should be set to a value roughly 80 times larger than the value returned by the `doc_estimate_dfs` function for a DITA map for which the output should be written to disk.

For more information on using Arbortext Publishing Engine for publishing, refer to the *Programmer's Guide to Arbortext Publishing Engine*, the *Programmer's Reference*, and the *Content Pipeline Guide*.

set bitmapdisplay

`set` bitmapdisplay= { on | off}

This command determines whether graphics and equations can be displayed in the Edit window or icons. This command overrides all previous `equationdisplay` and `graphicdisplay` option settings. `on` is the default.

is a synonym for `bitmapdisplay`.

set browserpath

`set` browserpath= *path*

This command sets the path and file name of the default HTML browser that Arbortext Editor will use to access URLs and preview documents on the Web.

You do not need to specify this command if you have a file association to a browser for HTML files, as Arbortext Editor automatically detects this association. However, you can use the `browserpath` option to override this file association.

You can set the path to a web browser in the **Tools ► Preferences** File Locations dialog box.

```
set browserpath=""C:\\Program Files\\Internet Explorer\\iexplore.exe""
```

set browserpreview

`set` browserpreview= { on | off}

When `browserpreview` is set to `on`, which is the default setting, the `.htm` file generated by **File ► Save as HTML** displays in a browser. The `.htm` file will not display unless you also have a file association for `.htm` files, the browser path preference specified in **Tools ► Preferences ► File Locations**, or the path specified by the `set browserpath` on page 759 command.

When set to `on`, `browserpreview` also selects the **View HTML** option in the publish dialog boxes.

The default setting is `on`.

set caretcolor

`set` caretcolor= { *colorname* | *#rgbspec*}

This command specifies the color of the editing cursor. The value is either a named color or an RGB specification preceded by `#`.

Examples (the following are equivalent):

```
set caretcolor=red
set caretcolor=#ff0000
```

set caretmovement

`set` caretmovement= { logical | visual}

This command sets the movement of the text cursor within documents that contain mixed left-to-right (LTR) and right-to-left (RTL) content. RTL editing is required for the Arabic and Hebrew locales.

The default is `logical`, which means cursor movements that are usually interpreted as moving to the left or right are instead interpreted as moving towards the start or end of a document. For example, pressing the left arrow key in RTL text will move the cursor towards the right. When you set this command to `visual`, the cursor moves to the literal left or right regardless of the locale.

set caretthickness

`set` caretthickness= *n*

This preference changes the thickness of the insertion cursor (caret). The value *n* must be in the range 1 to 10. If greater than 3, then the caret is drawn as a block cursor. The default value is 2.

set carettype

`set` carettype= { ibeam | block | line}

This command sets the style of the text caret in the Arbortext Editor window. There are three available values: *ibeam*, *block*, and *line*. If this command is not set, the default is *ibeam*.

set case

`set` case= { on | off}

The `case` option enables or disables case-sensitive searches. `off` is the default.

For example, when setting `case` to `off`, the search `find 'Basil'` finds both `Basil` and `basil`; however, when setting `case` to `on`, only `Basil` is found.

set catalogpath

`set` catalogpath=*dir1* [:*dirn*]

This command specifies the list of directories to search to locate `catalog` files. These files resolve public identifiers for both entities and document types. The initial setting is the value of the *APTCATPATH* environment variable if set. Otherwise `catalogpath` defaults to the *`Arbortext-path`*`\doctypes` directory.

If there is a catalog file in the *`Arbortext-path`*`\custom\doctypes` subdirectory at startup, the `\custom\doctypes` path is automatically prepended to the catalog path. If there are any subdirectories of the `\doctypes` directory that contain a catalog file, those subdirectories are also prepended to the

catalog path. Putting a catalog file in `\custom\doctypes` or in subdirectories of it makes them automatically available, avoiding additional steps to add them to the path.

If there is a catalog file in the *`Arbortext-path`*`\custom\composer` subdirectory at startup, the `\custom\composer` path is automatically prepended to the catalog path.

The *`Arbortext-path`*`\custom\doctypes` directory is also automatically included if you include `%D` in the path.

Example

```
set catalogpath="c:\\tmp2\\testdir;c:\\dauser;"
```

Multiple directories are delimited with semicolons, and the entire path must be enclosed in quotation marks.

You can use the `append_catalog_path` function on page 218 to update the catalog path.

set catalogwarnings

`set` catalogwarnings= { on | off}

This command enables or disables any warning messages when a catalog file is read. The default is `off`.

The command applies to TR9401 catalogs only.

set cgmprofile

`set` cgmprofile=*profile*

This preference is used in conjunction with the `graphicwebtransform` preference to indicate the type of CGM profile to which specified graphic file formats should be transformed during publishing to HTML outputs. When `cgm` is the target graphic file format for `graphicwebtransform`, Arbortext Editor uses the value of the `cgmprofile` preference to determine the CGM profile to use for the transformation. If `cgmprofile` is not set, then the graphics are transformed to the WebCGM 2.0 profile.

The following values for the CGM *profile* are supported:

- `WebCGM_2.0`
- `WebCGM_1.0`
- `ISO_8632_:_1999`
- `ATA_GREXCHANGE_V2.10`
- `ATA_GREXCHANGE_V2.9`

- ATA_GREXCHANGE_V2.8
- ATA_GREXCHANGE_V2.7
- ATA_GREXCHANGE_V2.6
- ATA_GREXCHANGE_V2.5
- ATA_GREXCHANGE_V2.4
- ATA_GREXCHANGE_V2.5/IsoDraw
- CALS_MIL-D-28003A
- SAE_J2008
- Model_Profile_(ISO_8632:1992/Am.1:1994)
- ISO_ISP_12071-1_(FCG11)_(Draft)
- ISO_ISP_12071-2_(FCG23)_(Draft)
- ISO_ISP_12071-3_(FCG32)_(Draft)
- ISO_ISP_12071-4_(FCG33)_(Draft)
- S1000D_V2.3
- S1000D_V2.2

Examples:

```
set cgmprofile=WebCGM_1.0
set cgmprofile=Model_Profile_(ISO_8632:1992/Am.1:1994)
```

set changetracking

`set` changetracking= { on | off}

Enables and disables change tracking for the current document. The state of this option is preserved in the document itself when it is saved. The default is `off`.

set changetrackingkeepdict

`set` changetrackingkeepdict= { on | off}

When this option is set to `off`, all change tracking markup is removed from a file when it is saved, if no change-tracked changes persist (i.e. all changes have been accepted or rejected). Any <atict:info> and <atict:user> elements in the document are deleted, and the *xmlns:atict* attribute that declares the namespace is removed from the top level document.

The default is `on`.

To restore the namespace declaration attribute, perform an undo action immediately after such a save.

The option is not saved as a preference.Set the option to `off` in a startup command file to maintain it as a permanent setting.

Note

If you have set custom change tracking colors, those settings will lost if `changetrackingkeepdict=off` and you save the file with no change-tracked changes in it.

set changetrackingmarkers

`set` changetrackingmarkers= { none | full}

This command controls how the change tracking markup tags are displayed in the edit window.

- `none` — Change tracking markup is not displayed.
- `full` — All change tracking markup is displayed.

Change tracking markup never appears in the Document Map pane. The default setting is `none`.

The following attributes are common to all change tracking markup:

Change Tracking Attributes

Attribute	Value	Description
ref	`number`	Uniquely identifies change tracking records.
user	`unique user identifier`	Identifies the user who made the changes.
time	`system time (sec)`	Represents the time of the change; recorded in seconds from 1/1/70, expressed as hh/mm/ss.
attr[1–9]	`any`	Used to customize change tracking applications.

For more detailed information on change tracking attributes, see http://www.arbortext.com/namespace/atict/change-tracking-markup-spec.html.

set changetrackingverbose

`set` changetrackingverbose= { on | off}

Controls the display of change tracking warnings. The default is `on`.

set charentdisplay

`set` charentdisplay= { on | off}

Changes the display of character entities in the edit window. If set to `off`, character entities are displayed as tags instead of glyphs or symbols. The default is `on`.

The `set entityinputconvert` option on page 790 must be set to `off` when opening the document for `set charentdisplay=off` to have an effect.

set charentmapfile

`set` charentmapfile= { *filename* | none}

This command changes the mapping table used by Arbortext Editor when converting non-ASCII characters into entity references on output. *filename* is the path name of an ASCII file containing two columns separated by white space; the first column specifies the entity name and the second specifies the character code in decimal, octal (with leading 0), or hexadecimal (with leading 0x). For example:

```
nbsp 160
iexcl 161
```

The default output mapping is given in the `isolat1.cf` file in the `lib` subdirectory of *`Arbortext-path`*. The actual translation table is built into Arbortext Editor so changing that file will have no effect unless the `charentmapfile` option is used.

`none` restores the default translation.

Examples

```
set charentmapfile=/user-path/charent.map
set charentmapfile=none
```

set cmdline

`set` cmdline= { on | off}

This option enables or disables the command line window. The default value is `off`. The default value can be changed through the **Show Command Line** setting in the **Window** category of the Arbortext Editor **Preferences** dialog box.

set cmsautoconnect

`set` cmsautoconnect= { on | off}

This option determines whether a repository adapter connect dialog box is automatically displayed when you attempt to open a file that is stored in the repository and no session is currently established with the repository. For example, you could select the file from the list of recently edited files on the Arbortext Editor **File** menu.

The default is `on`.

set colbreaktext

`set` colbreaktext=*string*

This command specifies the string, centered in the horizontal rule, that is used to denote a forced column break when breaks are displayed in the current tag display mode. The default is `Forced Column Break`.

Examples

```
set colbreaktext=""
set colbreaktext="New Column"
```

set columnrulerunit

`set` columnrulerunit= { in | cm | mm | pt | pc | pi}

This command sets the unit of measure for the Column Ruler. Available units are:

- `in` — inches (default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pc` — picas
- `pi` — picas

set compilesgml

`set` compilesgml= { on | off}

This command states whether a document type is SGML or XML. When Arbortext Architect loads a document type, it checks for the XML PI at the beginning of the document type. If the XML PI is present, the document type is

XML, and Arbortext Architect automatically sets `compilesgml` to `off`. If the XML PI is absent, the document type is SGML, and Arbortext Architect automatically sets `compilesgml` to `on`.

The value of `compilesgml` also determines whether the **SGML Application** or **XML Application** option is selected on the following menus:

- Arbortext Architect **Compile** menu
- Arbortext Architect DTD Editor **Tools** menu

If the you change the application value on these menus, the `compilesgml` value is also changed.

The default value of `compilesgml` is .

set composedcharactersubstitution

`set` composedcharactersubstitution= { on | off}

This option specifies whether the publishing engine will replace some ASCII characters with their typographical equivalents.

When `composedcharactersubstitution` is set to `on`, the characters get replaced by their typographical equivalents as described in the example. The replacements happen for elements that do not have the `allowCharacterSubstitution` control set to `off` in the `.dcf` file.

When `composedcharactersubstitution` is set to `off` (the default), ASCII characters are never replaced in any context.

Replacements never occur when publishing using XSL-FO.

```
39 (single quote ') -> 0x2019
96 (grave accent `) -> 0x2018
```

set composerpath

`set` composerpath=*dir1* [:*dirn*]

This command specifies a list of directories (each directory separated by a semicolon) in which publishing configuration files (`.ccf`) are located. These files are for customizing publishing processes. The default path is *`Arbortext-path`*`\composer`.

If there is an *`Arbortext-path`*`\custom\composer` subdirectory at startup, the `\custom\composer` path is automatically prepended to the path for `.ccf` files. Putting your `.ccf` files in the `\custom\composer` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

Arbortext Editor also looks at the value of `composerpath` to locate character entity substitution files used for HTML and XML output from publishing processes.

Example

```
set composerpath="c:\\user\\compfile.ccf;%D"
```

You can use the append_composer_path function on page 219 to update the publishing configuration file path.

set contextrules

`set` contextrules= { on | off}

This command enables or disables context checking. Note that if you have turned context to “off”, it does not reset when you leave and reenter a document. It remains off until you explicitly turn it on again. Because complex document structures can be difficult to maintain without context checking, PTC recommends that you leave context checking on. `on` is the default.

Note

Arbortext recommends that if you have been working with context checking off, you set `contextrules` to `on` before you save the document. When context checking is off, it is possible to create a document that is no longer valid SGML and that another system may be unable to read.

set contextwarnings

`set` contextwarnings= { on | off}

This command enables or disables the warning message about disabling context checking. `on` is the default.

set creoviewdownloaduri

`set` creoviewdownloaduri=*uri*

This command specifies a *uri* where the Creo View web installer (`.cab` file) is located. The *uri* parameter contains a URI (Uniform Resource Identifier) value.

Arbortext Editor writes the specified *uri* location to a generated HTML file when an intelligent graphic is published in the HTML file. This enables Internet Explorer to install Creo View when the HTML file is displayed in the browser. Creo View enables users to interact with the intelligent graphics in the HTML file.

If `creoviewdownloaduri` is not set or contains an empty value, you can still view intelligent graphics in a published HTML file. To support viewing intelligent graphics, Creo View must already be installed on the workstation where Internet Explorer is running.

Refer to the Creo View documentation for more information about Creo View.

Examples:

```
set creoviewdownloaduri=http://www.acme.com/creoview/creoviewexp1_PTC.cab
```

set creoviewfileformats

`set` creoviewfileformats=" [edz] [;iso] [;pvz] [;cgm] "

This command specifies both the graphic file formats that should be displayed using the embedded Creo View control in Arbortext Editor and the graphic types that should be published using Creo View. By default, the following types of graphics are displayed in the embedded control:

- EDZ (`.edz`)
- PVZ (`.pvz`)

You can also set the dialog box to also display CGM (`.cgm`) graphics. The arguments are the file extensions for the graphic types separated by semicolons. If you set the value to more than one graphic type, you must enclose the arguments in quotes. If you set the value of this option to an empty string, the Creo View control will not be used to display any graphics.

Publishing intelligent graphics using Creo View is also supported for all intelligent graphics types. Refer to the Creo View documentation for more information about Creo View.

Examples:

```
set creoviewfileformats=pvz
set creoviewfileformats="pvz;cgm"
```

set datamergepath

`set` datamergepath=*dir1* [:*dirn*]

This command specifies a list of directories in which data merge configuration files (`.dmf`) are located. These files are for customizing data merging operations. The default path is *`Arbortext-path`*`\datamerge`.

If there is an *`Arbortext-path`*`\custom\datamerge` subdirectory at startup, the `\custom\datamerge` path is automatically prepended to the path for data merge configuration files. Putting your `.dmf` files in the `\custom\datamerge` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

Refer to the *Customizer's Guide* for details on configuring your site to merge data.

Note

The capability to merge data is deprecated and not an encouraged best practice supported by PTC.

set datamergereadonly

`set` datamergereadonly= { on | off}

This command specifies the read-only status of merged data regions in documents. If set to `on` (the default), merged data regions are read-only.

Refer to the *Customizer's Guide* for details on configuring your site to merge data.

Note

The capability to merge data is deprecated and not an encouraged best practice supported by PTC.

set dcffile

`set` dcffile= { *name* | none}

This command specifies the document type configuration file (`.dcf`) to be used in place of the default `.dcf` file (*`doctype`*`.dcf`) for the current document's document type. *name* is the path and file name of the `.dcf` file, with or without a file extension. If you don't specify a file extension, Arbortext Editor searches for *name* with a `.dcf` file extension.

For example,

```
set dcffile="C:\doctypes\test\mydoctype.dcf"
```

set debugcomposition

`set` debugcomposition= { on | off}

When this option is `on`, Arbortext Editor creates a log directory for each publishing operation and stores the Event Log and intermediate files for the publishing request. After each publishing operation finishes, the log directory will be saved to a zip archive and then the directory will be deleted. The default is `off`.

When an application save directory is created (using **Tools ► Save Application**), the save directory will contain a file named `compose.zip`, which contains the zipped temporary directory for each publishing operation performed while `set debugcomposition` was set to `on`. When Arbortext Editor exits, the temporary files will be deleted.

For sites using Arbortext Publishing Engine to fulfill publishing requests from Arbortext Editor clients, refer to *Programmer's Guide to Arbortext Publishing Engine* and the Using Arbortext Publishing Engine for publishing documents online help topic.

set deepcontentsplitting

`set` deepcontentsplitting= { on | off | never}

This `set` option is used to enable and disable deep content splitting, which allows page breaks in table rows, algroups, and boxed regions. Deep content splitting is enabled using `on`, disabled using `off`. The default setting is `on`.

You can insert a `_deepsplit` tag pair to circumvent the `off` setting for `set deepcontentsplitting` using **Format ► Touchup ► Deepsplit Region**. To prevent deep content splitting overrides, set this command to `never`.

Note

Using line numbering with `deepcontentsplitting` may produce unexpected results. It is recommended that you do not use line numbering with `deepcontentsplitting`.

To disable deep content splitting in your document:

```
set deepcontentsplitting=off
```

set defaultfilter

`set` defaultfilter= *filterstring*

This command specifies the default list of file types to display in the **Filter** menu (or pull-down) on the **Open** dialog box.

filterstring specifies a list of file types separated by a vertical bar "|". There are two parts to each item, which are also separated by a vertical bar "|". The first part is the string to display in the menu or pop-up menu. The second part is the corresponding pattern to use to filter the files displayed in the dialog box. Multiple patterns may be specified by separating them with a semicolon.

Example

```
set defaultfilter="SGML Files|*.sgm;*.sgml|\
```

```
HTML Files|*.htm;*.html|\
Text Files|*.txt|All Files|*.*"
```

set defaultprintdpi

`set` defaultprintdpi= *value*

This command specifies the resolution value in images resulting from Arbortext Editor converting SVG and PVZ images for print and PDF output. *`value`* is a positive integer specifying the resolution in dpi (dots per inch). If `defaultprintdpi` is not set, Arbortext Editor assumes a value of `300`.

set defaultscreendpi

`set` defaultscreendpi= *value*

This command specifies the screen display resolution value for images resulting from Arbortext Editor converting SVG images to PNG images and for graphics files which do not have a resolution specified internally within the graphic. *`value`* is a positive integer specifying the resolution in dpi (dots per inch). If `defaultscreendpi` is not set, the default resolution values is `96`.

set deferotherreferenceupdates

`set` deferotherreferenceupdates= { on | off}

If this option is set to `on` Arbortext Editor will not update other references to entities, includes, or DMS objects which change if the other reference is in a document that is open in a window that doesn't have focus or is not open in any window. The reference without the focus will be updated when it gets the focus or if Arbortext Editor needs to update it (for example when saving the document or updating gentext for it).

If you open a file entity in its own window, turning this option on might make editing of the entity faster at the expense of not showing the most recent changes in the document that refers to the entity.

set deletespaces

`set` deletespaces= { on | off}

When this option is set to `on` (the default), Arbortext Editor will remove a space when a cut or delete operation results in two consecutive spaces. When this option is set to `off`, both spaces will remain. Setting the option to `off` is useful for developers that perform a delete and an insert in separate steps and do not want to lose the space.

set dialogdisplay

`set` dialogdisplay= { on | off}

If set to `on`, embedded XUI dialog boxes will be displayed in the Edit pane. If set to `off`, the XUI markup is displayed in the Edit pane. The default is `on`.

set dialogspath

`set` dialogspath=*dir1* [:*dirn*]

This command specifies a search path for dialog files that can be called from a custom application, such as one that uses the AOM `Application.createDialogFromFile` method. The default path is *`Arbortext-path`*`\lib\dialogs`. Each directory specified must be separated by a semicolon. You can specify the `%D` or `%H` symbolic parameters in the path list.

If an *`Arbortext-path`*`\custom\dialogs` subdirectory exists at startup, the `custom\dialogs` path is automatically prepended to the path for dialog files. Putting your custom dialog files in the `\custom\dialogs` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

Example

```
set dialogspath="c:\mydialogs;%D"
```

You can use the `append_dialogs_path` function on page 219 to update the dialog file path.

set diffattrmodcolor

`set` diffattrmodcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the font color used in the **Compare** window to display attribute modifications. If `inherit` is set, elements with changed attributes are displayed in the font color specified by the FOSI. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

Examples

```
set diffattrmodcolor=inherit
set diffattrmodcolor=red
set diffattrmodcolor=#ff0000
```

set diffattrmodname

`set` diffattrmodname=*name*

This command allows you to specify the name of the tag/processing instruction that encloses attribute modifications in the **Compare** window.

set diffdelcolor

`set` diffdelcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the color used in the **Compare** window to display text marked for deletion. If `inherit` is set, deleted elements are displayed in the color specified by the FOSI. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

Examples

```
set diffdelcolor=inherit
set diffdelcolor=red
set diffdelcolor=#ff0000
```

set diffdelname

`set` diffdelname=*name*

This command allows you to specify the name of the tag or processing instruction that encloses deleted text in the **Compare** window.

set diffencltype

`set` diffencltype= { pi | tag}

This command takes the value of `pi` if differences in the **Compare** window are to be enclosed within processing instructions. If `tag` is specified, the values of `diffinsname`, `diffdelname`, and `diffattrmodname` must all be valid elements of the DTD for the document being compared. Also, these elements must be able to appear anywhere in the document. The default is `pi`.

set diffentities

`set` diffentities= { on | off}

If `on` is specified, entities will be expanded in the **Compare** window and the entity contents compared between the source and target documents. In this case, the expanded content appears in the **Compare** window. If the Compare document is saved, then the expanded entity content is flattened and saved in the Compare document.

If `off` is specified, the entity references are compared instead of their content. The default is `off`.

set diffignoreattrs

`set` diffignoreattrs= { on | off}

When this command is set to `on` (the default), modifications to attributes are not reported in the **Compare** window.

set diffincludes

`set` diffincludes= { on | off}

If `on` is specified, files included using XML inclusions will be expanded in the **Compare** window and the contents will be compared between the source and target documents. In this case, the expanded content appears in the **Compare** window. If the Compare document is saved, then the expanded included content is flattened and saved in the Compare document.

If `off` is specified, XML inclusion references are compared instead of their content. The default is `off`.

set diffinscolor

`set` diffinscolor= { inherit | *colorname* | *#rgbspec*}

This command determines the color used in the **Compare** window to display inserted text. If `inherit` is set, inserted text is displayed in the color specified by the FOSI. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

Examples

```
set diffinscolor=inherit
set diffinscolor=red
set diffinscolor=#ff0000
```

set diffinsname

`set` diffinsname=*name*

This command allows you to specify the name of the tag or processing instruction that encloses inserted text in the **Compare** window.

set diffmemory

`set` diffmemory= *n*

This option specifies the maximum memory allocation used by Arbortext Editor during document comparison processing. The value is in megabytes. The default value is 256. If memory allocation is insufficient, you'll get an error message instructing you to set a larger value.

set diffstrikethrough

`set` diffstrikethrough= { on | off}

This command controls the appearance of deleted text in the **Compare** window. When `diffstrikethrough` is set to `on` (the default), deleted text appears in the assigned color and in strike-though characters. When `diffstrikethrough` is set to `off`, deleted text appears in the assigned color and in regular characters.

set diffunderline

`set` diffunderline= { on | off}

This command controls the appearance of inserted text in the **Compare** window. When set to `on` (the default), inserted text appears in the assigned color and in underlined characters. When `diffunderline` is set to `off`, inserted text appears in the assigned color and in regular characters.

set ditacheckreferences

`set` ditacheckreferences= { on | off}

This option determines whether the Arbortext Editor enhanced completeness checking for DITA documents opens referenced documents and performs the enhanced check on them as well as the current document. Set this option to `on` to check references during enhanced completeness checking and to `off` to just check the current document. The default is `on`. This option has global scope.

set ditaexpectedformats

`set` ditaexpectedformats=dita [;ditamap] [;xml] [;html] [;pdf] [;txt] [;sgm] [;sgml]

This option contains a semicolon-separated list of possible values for the DITA *format* attribute. The list is used during enhanced completeness checking for DITA documents when checking that the attribute has a valid value. The following default format values are used:

- `dita`
- `ditamap`

- `xml`
- `html`
- `pdf`
- `txt`
- `sgm`
- `sgml`

This option has global scope.

set ditahideids

`set` ditahideids= { on | off}

This option determines whether the **ID** option is displayed on the **Resource Manager** dialog's **New Topic** tab and whether IDs are displayed in the **Resource Manager** browser. Set this option to `on` to hide DITA IDs and to `off` to display IDs. The default is `on`. This option has global scope.

set ditaincludecommentsinrds

`set` ditaincludecommentsinrds= { none | all | map | topic}

This option specifies whether authored comments are included in the Resolved Document for Styling (RDS).

- *none* (default) — no comments are included
- *all* — comments from maps and topics
- *map* — comments from map only
- *topic* — comments from topics only

Note

The RDS is normally an intermediate output. Further processing may be required to have this information included in final output.

set ditaincludemapsinrde

`set` ditaincludemapsinrde= { on | off}

This option determines whether the markup for DITA maps is included in an automatically generated resolved document for editing. If you use a custom document type for the resolved document for editing that includes DITA maps, the

option also determines whether maps are included in the resolved document in this case. If you use a custom document type that does not include DITA maps, the option has no effect.

Set this option to `on` to include DITA map markup and to `off` to not include map markup. The default is `off`. This option has global scope.

set ditainsertallwarnings

`set` ditainsertallwarnings= { off | *n*}

This option determines whether Arbortext Editor displays a warning dialog box when more than a designated number of topic references are being inserted into a DITA map in a single operation. The default is `20`. Set this option to `off` to prevent a warning dialog box from being displayed. Set this option to a different number to change the default threshold. Setting this option to `0` has the same effect as setting it to `off`. This option has global scope.

Examples

```
set ditainsertallwarnings=off
set ditainsertallwarnings=30
```

set ditakeybaselist

`set` ditakeybaselist=*URI* [;*URI*]

This option contains a semicolon-separated list of URIs of DITA maps containing key definitions. The keys in those maps are used to populate the **Insert Key Reference** dialog box and are also used to populate the **Keyref** field on the various **Resource Manager** tabs and versions when an object with a matching key is selected. This preference can be updated for a specific document using the **Add/Remove Map** dialog box. However, changes made to this preference are only persistent if the value is set from the **Advanced Preferences** dialog box or by using the command itself.

This option has document scope and does persist across sessions.

set ditakeycontext

`set` ditakeycontext=*URI*

This option contains the URI of the source of key definitions for key references in the current document. You can also set this option explicitly or Arbortext Editor sets it automatically in some cases. For example, when a user opens a topic from within a map, Arbortext Editor automatically sets that topic's `ditakeycontext` value to the URI of the map. The elements in the topic with *keyref* attributes referencing keys will then use the key definitions defined in that map.

This map is also used, in addition to the maps set in the `ditakeybaselist` preference, to populate the **Keyref** field on the various **Resource Manager** tabs and versions. This option has document scope and does not persist across sessions.

set ditakeyreffallback

`set` ditakeyreffallback= { on | off}

When entering key references via the **Insert Key Reference** dialog box, this option determines whether an *@href* and other attributes will also be added to the reference elements that are added, such as xrefs and links. The fallback markup is used when processing the topic during publishing. If the key reference cannot be resolved, the fallback markup is used to resolve the reference.

The default is `on`.

set ditakeynamequalifier

`set` ditakeynamequalifier=*qualifier*

This option contains a string that is automatically added as a prefix or suffix to the key names you enter from the **Resource Manager**'s **Key Definition** tab. This helps ensure that your key names are unique. This option has document scope.

Whether the qualifier is added as a prefix or suffix is determined by how the qualifier string begins or ends. The following rules apply for key name qualifiers:

- If a qualifier begins with a `.` or `-`, the qualifier is a suffix.
- If a qualifier ends with a `.` or `-`, the qualifier is a prefix.
- if a qualifier neither ends nor begins with a `.` or `-`, the qualifier is a suffix.

It is recommended that you use a suffix as the name qualifier and that you use a `.` to separate the parts of the qualifier.

Examples

If you set `ditakeynamequalifier` to the following value:
```
set ditakeynamequalifier=.dita.acme.com
```

Your key names will have a name qualifier suffix:
```
keyname.dita.acme.com
```

If you set `ditakeynamequalifier` to the following value:
```
set ditakeynamequalifier=com.dita.acme.
```

Your key names will have a name qualifier prefix:
```
com.dita.acme.keyname
```

set ditakeyrefui

`set` ditakeyrefui= { on | off}

This option determines whether the user interface options for keys and key reference support are displayed in the Arbortext Editor interface. Set this option to `on` to display the key reference user interface and to `off` to hide the interface. The default is `on`. This option has global scope.

Refer to Using keys and key references in the Arbortext Editor help for details about the key reference user interface options.

set ditanewfilelang

`set` ditanewfilelang= { *lang* | none}

This option determines the default language to which Arbortext Editor sets the value of the *xml:lang* attribute on the root tag of DITA maps and topics created from templates. If this preference is not set to a value, then the default language of the stylesheet associated with the document is used. If the default language of the stylesheet is `en`, then the system locale is used for the language.

If you set this preference to `none`, then no *xml:lang* value is set on new documents. However, this is not recommended. This option has global scope.

Examples

```
set ditanewfilelang=de
set ditanewfilelang=none
```

set ditapath

`set` ditapath=*d1* [;*dn*]

This command specifies a list of directories to search for content referenced by DITA documents when their references do not specify either an absolute path name or a path name relative to the current document directory. Separate each directory in the DITA references path by a semicolon.

You can use symbolic parameters in the DITA references path. The DITA references resolved by this variable include topic references, content references, links, cross references, and any other DITA element that references content. The only exception to this is graphic references, as they are resolved using the `set graphicspath` on page 828 option.

The initial setting is the value of the *APTDITAPATH* environment variable, if set. Otherwise, it defaults to the *`Arbortext-path`*`\ditarefs` directory. If a `ditarefs` subdirectory exists in either the *`Arbortext-path`*`\application` or *`Arbortext-path`*`\custom` directory, that subdirectory is also part of the default DITA references path.

The base URI for the current element and the document directory, if different from the base URI, are included by default before any directories on the DITA references path to resolve relative references. When any `set ditapath` directory parameter is set to a value of `%^`, the document directory and the base URI for the current element will not be included when searching for DITA references. Only the paths specified will be searched.

The DITA references path can also be set in the **File Locations** category of the **Tools ► Preferences** dialog box.

Examples

```
set ditapath="c:\\Program Files\\Arbortext\\documents\\dita;%^;c:\\DITA\\content"
```

A list of multiple directories must be enclosed in quotation marks.

You can use the `append_dita_path` function on page 220 to update the DITA references path.

set ditareltableautoinsert

`set` ditareltableautoinsert= { on | off}

This option determines whether a topic reference entered into a DITA relationship table through drag-and-drop is automatically put into the table column that matches the DITA topic type. Set this option to `on` to automatically put the topic reference into the matching table column and to `off` to put the topic reference into the column where the reference is dropped. The default is `on`. This option has global scope.

set ditasynctabs

`set` ditasynctabs= { on | off}

This option determines whether the **Synchronize Location Across Tabs** choice is on or off by default in the **Resource Manager** menus. The default is `on`. When set to on, the location you browse to on a **Resource Manager** tab is maintained when you switch to another tab or open a new **Resource Manager** dialog box. This option has global scope.

set ditatextkeyrefs

`set` ditatextkeyrefs= { on | off}

This option determines whether non-linking DITA elements are included in the **Insert Key Reference** dialog box's **Insert** option. The default is `off`. When set to `on`, non-linking elements that allow a key reference are included in the **Insert** option. This option has global scope.

set ditausenewrds

`set` ditausenewrds= { on | off}

This option specifies whether to use a new format Resolved Document for Styling (RDS) that was introduced in release 6.1 M030 for styling and publishing of DITA content.

The new format RDS displays certain differences to the old format:

- No `ReferredTopics` section
- Does not contain all the metadata from topics in the `FlattenedMap` section
- Does not include some of the options controlled by `dita.acl` (control of certain customizations)

The new format RDS is used by default. Set the command to `off` to use the old format RDS.

set ditavaldebug

`set` ditavaldebug= { on | off}

This option determines whether debugging information is displayed in the **DITAVAL Preview Window** when a DITAVAL file is tested. Set this option to `on` to display debugging information. The default is `off`. This option has global scope.

When debugging is turned on, the following information is included in the **DITAVAL Preview Window** when testing a DITAVAL file:

- A comment is inserted at the beginning of the document listing all of the DITAVAL files used to process the document.
- A comment is inserted before any affected tags describing why the content was included, excluded, or flagged. The comment includes the DITAVAL file that the rule came from with a description of the rule.
- All `exclude` rules are flagged in red with a yellow background and strikethrough text.

set docmapcurrenttag

`set` docmapcurrenttag= { on | off}

This option controls how the `oid_current_tag()` and `current_tag_name()` ACL functions and the *$tagname* predefined variable determine the "current tag" in the Document Map view. It is provided for backward compatibility with certain pre-Adept 7.0 ACL scripts.

If this option is set to `on` (the default), Arbortext Editor determines the "current tag" per the following rules:

- If the cursor is anywhere in the Edit view, the current tag is always the tag to the left of the cursor.
- If the cursor is at the start of a line in the Document Map view (that is, between the element icon and the element name), the current tag is the element to the right of the cursor.
- If the cursor is anywhere else in the Document Map, the current tag is the element to the left of the cursor.

If the option is set to `off`, then the current tag is always the tag to the left of the cursor in both the Document Map and Edit views.

If you have an ACL script from a pre-7.0 version of Adept that uses the current tag, you may want to add the `set docmapcurrenttag=off` command to the beginning of the script. This will guarantee that the "current tag" will always be to the left of the cursor, even if the cursor is at the start of a line in the Document Map view. When your script completes, it should execute the `set docmapcurrenttag=on` command to restore the default value.

If your script relies on a user's interaction with Arbortext Editor, you may wish to leave `docmapcurrenttag` set to its default value of `on`.

For example, assume that your script makes specific modifications to element attributes for the current element (tag). If a user executes your script using a menu command or control-key, you are relying on them to position the cursor such that they modify the attributes for the desired element. In this case, you would want to exploit the intuitive nature of "current tag" determination in the Document Map view.

Note

The `modify_tag`, `change_tag`, and `delete_tag` commands operate on the current tag, so they are affected by this option setting.

set docmapendtags

`set` docmapendtags= { full | partial | none}

Use this set command to control the display of end tags in the Document Map view. This window-specific setting is retained whether a Document Map view is in the current window or not. If the Document Map is currently disabled, this command will have not visible affect until the Document Map is enabled in that window.

The valid settings are:

- `full` — Shows both the end tag icon and element name.
- `partial` — Shows the end tag icon only.
- `none` — Shows neither the end tag icon nor the element name.

set docmapgentext

`set` docmapgentext= { on | off}

This command determines if generated text is displayed in the Document Map pane within the Edit window. When set to `on`, the text is displayed, when set to `off` it is not. The default is `on`.

When displayed, the generated text will update according to the setting of the `gentextautoupdate` option.

set docmaphighlight

`set` docmaphighlight= { *colorname* | *#rgbspec*}

This command specifies the color value used to highlight the Document Map line containing the cursor. The value is either a named color or an RGB specification preceded by `#`. The default is no highlighting. Set the value of this option to `default` to remove any existing highlighting.

The `docmaphighlight` command is an advanced preference with global scope. When you edit this preference, click on the down arrow to display the Color palette.

Examples

```
set docmaphighlight=blue
set docmaphighlight=#dd00dd
```

set docmapmode

`set` docmapmode= { on | off}

If set to `on`, the current window is split into two side by side panes with the Document Map in the left view and the document in the right view. If set to `off`, the current window consists of a single view containing the document (not the Document Map).

The position of the split between the windows is set by the `docmappercent` option. To turn the Document Map on/off in the current view, use the `docmapview` option.

Note

This command is deprecated beginning in Arbortext Editor 5.3. Use the `set viewmode` command to change the current view.

set docmappastecaret

`set` docmappastecaret= { on | off}

This option determines the position of the caret (cursor) in the Document Map view after a `paste`, `insert_string` , or `insert_entity` command or `insert()` function completes. This option is provided for backward compatibility with certain pre-Adept 7.0 ACL scripts.

In the Edit view, the caret is always positioned immediately after the pasted region.

In the Document Map view, the caret is placed immediately before the pasted region if the `docmappastecaret` option is set to `on` (the default). This default setting in the Document Map minimizes horizontal scrolling after a paste or insert operation. If `docmappastecaret` is set to `off`, the caret is positioned at the end of the region after the paste or insert.

If you have an ACL script from a pre-7.0 version of Adept that relies on the cursor being placed after pasted regions, you should set the `set docmappastecaret=off` command at the beginning of the script. This will guarantee that the “current tag” will always be to the left of the cursor, even if the cursor is at the start of a line in the Document Map view. When your script completes, it should execute the `set docmappastecaret=on` command to restore the default value.

set docmappercent

`set` docmappercent= *n*

This option determines the percentage of the window devoted to the Document Map or Column view display when it is activated. The value *n* must be in the range `0` to `100`. If `0`, the Document Map is hidden, if `100` the Document Map will occupy the entire window. The default is `33` percent.

n is a number, excluding a percent (`%`) symbol.

set docmapshowattrs

`set` docmapshowattrs= { on | off}

This option controls the display of attributes in the Document Map or Column view. This window-specific setting is retained whether a Document Map or Column view is in the current window or not. If the Document Map or Column view is currently disabled, this command will have no visible effect until the view is enabled in that window.

The current valid settings are:

- `on` — Displays each element attribute in the Document Map or Column view as a single non-wrapped line.
- `off` — Displays no element attributes in the Document Map or Column view. This is the default setting.

This setting is the same as choosing the **View ► Expand Attributes** menu choice.

set docmapside

`set` docmapside= { left | right | top | bottom}

This option determines where Arbortext Editor will place the Document Map or Column view in a split window view. The default is `left` in most locales. However, in the Arabic and Hebrew locales where right-to-left editing is required, the default is `right`.

set docmapsync

`set` docmapsync= { on | off}

This option determines whether Arbortext Editor will automatically synchronize when a document is displayed in both an Edit view and a Document Map or Column view. This command has the same affect as choosing the **Synchronized Views** preference in the **View** category of the **Preferences** dialog box.

set docmaptextdisplay

`set` docmaptextdisplay= { none | nowrap | wrap}

This command controls the appearance of the content text displayed in the Document Map view. This window-specific setting is retained whether a Document Map view is in the current window or not. If the Document Map is currently disabled, this command will have no visible effect until the Document Map is enabled in that window.

Valid settings are:

- `none` — Displays no content text in the Document Map view (Displays structure only).
- `nowrap` — Displays content text in the Document Map view as single non-wrapped lines. Using this setting is the same as choosing the **View⇒Document Map⇒Line Text in Map** menu option. You can control how your content text is lined up with the `docmapusetabs` option.
- `wrap` — Displays content text in the Document Map view text-wrapped lines. The width of the text-wrap column is controlled by the `docmapwidth` option. Using this setting is the same as choosing the **View⇒Document Map⇒Wrap Text in Map** menu option. You can control how your content text is lined up with the `docmapusetabs` option.

set docmapusetabs

`set` docmapusetabs= { on | off}

This command controls how your Document Map content text displays when using the `nowrap` and `wrap` settings for `docmaptextdisplay`. This window-specific setting is retained whether a Document Map view is in the current window or not. If the Document Map is currently disabled, this command will have not visible affect until the Document Map is enabled in that window.

Valid settings are:

- `on` — Lines up the content text in hierarchical fashion.
- `off` — Does not line up content text; content text starts a fixed distance from the end of the element name.

set docmapview

`set` docmapview= { on | off}

When set to `on`, the current view changes to a Document Map.

To split the window and show the Document Map in the left view, use the `docmapmode` option.

Note

This command is deprecated beginning in Arbortext Editor 5.3. Use the `set view` command to change the current view.

set docmapwrapwidth

`set` docmapwrapwidth= *width*

This command controls how your Document Map content text lines up when using the `wrap` setting for `docmaptextdisplay`. This window-specific setting is retained whether a Document Map view is in the current window or not. If the Document Map is currently disabled, this command will have no visible affect until the Document Map is enabled in that window.

Enter the desired column width for the content text in *width* argument. (For example, `8cm`, `5in`, and so on.) The default units is inches. Settings that would result in a content text column of less than two (2) inches are ignored.

set doctypecachesize

`set` doctypecachesize= *size*

This command controls the size of an internal cache that retains document type information and associated stylesheets during a Arbortext Editor or Arbortext Publishing Engine session. The first time a document is opened, the associated document type and stylesheet are compiled (if needed) and loaded. When the `doctypecachesize` option is enabled, an empty document for the same document type and stylesheet is created and loaded in the cache. This causes Arbortext Editor and Arbortext Publishing Engine to retain the associated document type and stylesheet, improving performance when another document of the same type is opened.

The default cache size is `20`, which is the number of document type and stylesheet pairs retained in the cache. If the cache size is exceeded, the oldest document type and stylesheet pair is removed from the cache. You can set the cache to a value between `0` and `50`. Setting the cache size to `0` clears and disables the cache, causing a document's document type and stylesheet to be loaded every time you open a document.

set documenttypewarnings

`set` documenttypewarnings= { on | off}

Specifies whether warnings should be emitted when parsing a DTD or Schema.

- `on` — The default when running Arbortext Architect. Warnings are reported.
- `off` — The default when running Arbortext Editor. Warnings are suppressed.

The potential warnings include:

- A notation already exists
- An attribute already exists

- Contradictory encoding
- Undeclared elements referenced in a content model
- Undeclared elements referenced in an attribute

set editduringformat

`set` editduringformat= { yes | no | prompt}

This option determines whether Arbortext Editor allows the user to continue editing a document while it's being formatted for a publishing request. The settings are:

- `yes` — Allow editing.
- `no` — Do not allow editing. When set to `no`, Arbortext Editor reports the document is read only.
- `prompt` — Prompt the user to decide whether to allow editing. This is the default setting.

set editfontpercent

`set` editfontpercent= *n*

This command increases or decreases the point sizes of all fonts in the Edit window to the percentage specified, subject to the availability of fonts. By default, fonts are displayed at 92% of their point size. Fonts cannot be displayed at less than 40% of their point size.

n is a number, excluding a percent (`%`) symbol.

set editselectionrecordlength

`set` editselectionrecordlength= { *n* | infinity}

The `editselectionrecordlength` enables you to set the record length (line breaks) of the part of a document selected in Arbortext Editor. Only the selected part of a document is affected by this command. The record length is set in the XML or SGML source. If you set the record length to `infinity`, no line breaks are included in the source of the selection.

Examples

```
set editselectionrecordlength=70
set editselectionrecordlength=infinity
```

set emptyelementswarnings

`set` emptyelementswarnings= { on | off}

The `emptyelementswarnings` option controls whether Completeness Checking tests for the presence of empty elements. Its default value is `off`.

set encodemediafilenames

`set` encodemediafilenames= { on | off}

The `encodemediafilenames` option controls whether filenames of media objects, for example graphics, may be encoded to remove non-ASCII characters.

Encoded filenames are longer than their original versions and may exceed Windows maximum file path lengths. This can cause issues with copy and move actions during publishing.

The default value is `on`.

Note

A publish to RTF action may have issues with non-ASCII filenames. PTC recommends that you enable the **Embed graphics** option in your Arbortext Styler stylesheet if encoding is disabled for RTF publishing.

set entityinputconvert

`set` entityinputconvert= { on | off}

The `entityinputconvert` option controls how Arbortext Editor stores character entities in memory when a document is opened, or when character entities are pasted or inserted in a document. When `entityinputconvert` is set to `on`, which is the default, Arbortext Editor converts character entity references to characters. This method saves memory space.

When `entityinputconvert` is set to `off`, Arbortext Editor stores character entities in a special data structure, which can be displayed in the Edit window as a character. This preserves the identity of the character entity, so that it can be written out in the same way that it was read in.

set entitylist

`set` entitylist= *c1* [*cn*]

This command specifies the list of character entities to display in the tool bar **Insert Symbol** pulldown list. The entities are separated by a space character. The list can contain numeric references as well as character entity names. When a numeric reference is specified, the character will be shown in the **Insert Symbol**

pulldown list if the character is defined in `charent.cf`. If the character is not defined in `charent.cf`, the hexadecimal value will be listed instead. If `entitylist` is set to a null string, no entities are displayed.

Entities which are not valid for a given document are ignored, although they remain in the option list. Each new document is initialized to this list, which may be overridden by a processing instruction in the file.

Examples:

```
set entitylist="alpha beta copy trade"
set entitylist="alpha &#x2126; #255"
set entitylist=""
```

set entityoutputconvert

`set` entityoutputconvert= { on | off}

The `entityoutputconvert` option controls how Arbortext Editor writes out character entities it stores in special data structures. It works with the `entityinputconvert` and `writenonasciichar` options.

If `entityoutputconvert` is set to `off`, which is the default, Arbortext Editor writes out character entities in the same format as they were read in.

If `entityoutputconvert` is set to `on`, Arbortext Editor writes out character entities as characters, numeric character references, or entity references, depending on the value of the `writenonasciichar` option.

Note

The `write` command `-eoc` and `-noeoc` options override the `entityoutputconvert` option settings.

set entitypath

`set` entitypath= *d1* [:*dn*]

This command specifies a list of directories to search for entities that do not specify an absolute path name or a path name relative to the current directory (each directory separated by a semicolon). The initial setting is the value of the *APTENTPATH* environment variable if set. Otherwise, it defaults to the `entities` subdirectory of *`Arbortext-path`*. This path option supports file entities as well. Use `%D` to refer to the *`Arbortext-path`*`\entities` subdirectory. Use `%B` to refer to directories relative to the current document's directory.

The document's parent directory and current working directory are included by default in the directories searched for entities. When any `set entitypath` directory parameter is set to a value of `%^`, the parent document directory and the current working directory will **not** be included when searching for entities. Only the paths specified will be searched.

If there is an *`Arbortext-path`*`\custom\entities` subdirectory at startup, the `\custom\entities` path is automatically prepended to the entities path. Putting your entity files in the `\custom\entities` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

The *`Arbortext-path`*`\custom\entities` directory is also automatically included if you use `%D` in the path.

Examples:

```
set entitypath="c:\\Program Files\\Arbortext\\editor\\entities;%^;
 c:\\Arbortext\\editor\\packages\\entities"
```

The list of multiple directories must be enclosed in quotation marks.

You can use the `append_entity_path` function on page 220 to update the entities path.

set entityscan

`set` entityscan= { on | off}

This command controls the default for the *entityscan* parameter on the `spell` and `find` commands and the default setting for the **Search Entities** check box on the **Find** and **Spelling** dialog boxes.

If `entityscan` is set `on`, file entities will be scanned during spelling check and find operations. You can override this default setting by using the `-es` or `-noes` options with the `spell` and `find` ACL commands, or by changing the **Search Entities** check box in the **Find** and **Spelling** dialog boxes.

The default is `on`.

set epubstylesheet

`set` epubstylesheet= { *name* | none}

This command specifies the stylesheet to be used as the default for the **Publish ▸ For EPUB** option on the **File** menu.

name is the path and file name of a `.style` or `.xsl` stylesheet file. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, and then a `.xsl` file.

The stylesheet ID processing instruction must specify `epub` as the publishing type.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:
  ```
  set epubstylesheet=D:\ArbortextUser\axdocbook.style
  ```
- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.
  ```
  set epubstylesheet=axdocbook.style
  ```

If `epubstylesheet` is set to `none` or is not set, Arbortext Editor and Arbortext Publishing Engine use the Editor/Default stylesheet.

Examples

```
set epubstylesheet=memo2
set epubstylesheet=~/mydocs/memo/memo2.style
set epubstylesheet=http://www.some_site.com/examples/techman.xsl
```

set epubinstalldir

`set` epubinstalldir= *path*

This command specifies the directory in which Calibre is installed.

A valid Calibre install directory must contain the files `ebook-convert.exe` and `ebook-viewer.exe`.

set equationdisplay

`set` equationdisplay= { on | off}

This command determines whether equations will be displayed in the Edit window. is the default.

is a synonym for `equationdisplay`.

set expandinclusions

This option determines whether XML inclusions will be expanded when a document is opened. The default value is `true`. If the value is `false`, when opening a document to be displayed, XML inclusions will not be expanded and the inclusion markup will appear in the document. (In Arbortext Editor, you can choose **View ► Included Object** to individually expand the inclusions.

set expressions

`set` expressions= { on | off}

This option allows or disallows the use of regular expressions when specifying a string with the `find` on page 646, `substitute` on page 715, `caret` on page 621, and `window` on page 727 commands without the `-e` modifier. The default is `off`.

This option also specifies the default values for matching patterns with the **Find/Replace** dialog box.

set extendselection

`set` extendselection= { on | off}

When this option is set to `on`, it turns on extend selection mode and displays **EXT** in the status bar. When enabled, any cursor movement, such as using the arrow keys, causes the highlighted region to be extended to the new cursor position. This mode remains in effect until `extendselection` is set to `off`, which ends the selection. The default is `off`.

set featureChangetracking

`set` featureChangetracking= { on | off}

This command enables (`on`) and disables (`off`) the user interface and underlying program for the change tracking feature. This option can only be changed in the `installprefs.acl` file that is loaded at program startup. The default `installprefs.acl` file is located in the *`Arbortext-path`*`\lib` directory of your Arbortext Editor installation directory.

If you create a custom `installprefs.acl`, copy the default *`Arbortext-path`*`\lib\installprefs.acl` file, and set the `set featureChangetracking` command.

You can make a custom `installprefs.acl` file automatically available by putting it in the user's *`Arbortext-path`*`\custom\lib` directory to load it automatically upon startup (avoiding manual steps to add it to the path).

set featureDMS

`set` featureDMS= { on | off}

This command enables (`on`) and disables (`off`) the user interface and underlying program for the PTC Server connection. This command must be included in the `installprefs.acl` file that is loaded at program startup. The default `installprefs.acl` file is located in the *`Arbortext-path`*`/lib` directory.

You can make a custom `installprefs.acl` file automatically available by putting it in the user's *`Arbortext-path`*`\custom\lib` directory to load it automatically upon startup (avoiding manual steps to add it to the path).

set featureImportExport

`set` featureImportExport= { on | off}

This command enables (`on`) and disables (`off`) Arbortext Import/Export. This option can only be changed in the `installprefs.acl` file that is loaded at program startup. The default `installprefs.acl` file is located in the *`Arbortext-path`*`\lib` directory of your Arbortext Editor installation directory.

If you create a custom `installprefs.acl`, copy the default *`Arbortext-path`*`\lib\installprefs.acl` file, and set the `set featureImportExport` command.

You can make a custom `installprefs.acl` file automatically available by putting it in the user's *`Arbortext-path`*`\custom\lib` directory to load it automatically upon startup (avoiding manual steps to add it to the path).

Note

Setting this feature to `on` does not circumvent your requirement to purchase a license for this feature.

This command is not supported for the compact installation of Arbortext Editor.

set featurePrintPublishing

`set` featurePrintPublishing= { on | off}

This command enables (`on`) and disables (`off`) the user interface and underlying program for the Print Composer product option. This option can only be changed in the `installprefs.acl` file that is loaded at program startup. The default `installprefs.acl` file is located in the *`Arbortext-path`*`\lib` directory of your Arbortext Editor installation directory.

If you create a custom `installprefs.acl`, copy the default *`Arbortext-path`*`\lib\installprefs.acl` file, and set the `set featurePrintPublishing` command.

You can make a custom `installprefs.acl` file automatically available by putting it in the user's *`Arbortext-path`*`\custom\lib` directory to load it automatically upon startup (avoiding manual steps to add it to the path).

Note

Setting this feature to `on` does not circumvent your requirement to purchase a license for this feature.

This command is not supported for the compact installation of Arbortext Editor.

set featureWebPublishing

`set` featureWebPublishing= { on | off}

This command enables (`on`) and disables (`off`) the user interface and underlying program for the Web/Wireless Composer product option. This option can only be changed in the `installprefs.acl` file that is loaded at program startup. The default `installprefs.acl` file is located in the *`Arbortext-path`*`\lib` directory of your Arbortext Editor installation directory.

If you create a custom `installprefs.acl`, copy the default *`Arbortext-path`*`\lib\installprefs.acl` file, and set the `set featureWebPublishing` command.

You can make a custom `installprefs.acl` file automatically available by putting it in the user's *`Arbortext-path`*`\custom\lib` directory to load it automatically upon startup (avoiding manual steps to add it to the path).

Note

Setting this feature to `on` does not circumvent your requirement to purchase a license for this feature.

set fileentityfontcolor

`set` fileentityfontcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the font color used to display file entities in the Edit window. If `inherit` is set, the file entities are displayed the same as the text of the Edit window document. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

You can determine whether Arbortext Editor ignores this setting by using the `set usecolorsettings` on page 912 option.

Examples

```
set fileentityfontcolor=inherit
set fileentityfontcolor=red
set fileentityfontcolor=#ff0000
```

set fileentitymarkers

`set` fileentitymarkers= { none | full | partial | default}

This command determines the display of file entity markup in the Edit window document.

- `default` — File entity markup takes on the same setting as the Edit window document tag display
- `none` — (The default.) No file entity tags or markers are displayed.
- `full` — All file entity markup tags and markers are displayed.
- `partial` — Only the file entity markers around the file entity are displayed.
- `icon` — Only icons and fully-collapsed entity content are displayed.
- `hide` — Only file entity icons are displayed.

set filelist

`set` filelist=*f1* [*f1*,*label*] [;*f2*;*f3*;…;*f8*] [;*f2*,*label*;*f3*,*label*;…;*f8*,*label*]

This command specifies the list of file names to display in the **File** menu as the most recently opened files. You can specify up to eight file names separated by the path list separator character (a semicolon). Each file name should be an absolute path name. You can supply an optional label to display in the list of file names instead of the path to the file. The label must follow the path name and be separated from the path name by a comma (',').

Arbortext Editor automatically updates this list each time a new document is opened. If set to a null string, no files will be listed in the **File** menu.

Examples

```
set filelist="C:\docs\art.xml;C:\packages\_main.acl"
set filelist="C:\process\initial.xml,Initial Process;
 C:\process\final.xml,Final Process"
set filelist=""
```

set filereference

`set` filereference= { xinclude | entity}

This set command specifies the method that will be used for inserting file references. The default value is `xinclude` indicating to insert references as XML inclusions. Set the value to `entity` to insert file entity references.

set fmtfaultfloat

`set` fmtfaultfloat= { ignore | report | stopafterpass | stopatonce}

This command determines how a float formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultgraphicoverset

`set` fmtfaultgraphicoverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a graphic overset (horizontal) formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaulthardkeeps

`set` fmtfaulthardkeeps= { ignore | report | stopafterpass | stopatonce}

This command determines how a hard keeps formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaulthdrftroverset

`set` fmtfaulthdrftroverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a header or footer overset formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultlineoverset

`set` fmtfaultlineoverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a line overset formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultlineunderfull

`set` fmtfaultlineunderfull= { ignore | report | stopafterpass | stopatonce}

This command determines how a line underfull formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — (the default) Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — Log the fault and display the fault in the **Formatter Messages** window.

- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultoverstretched

`set` fmtfaultoverstretched= { ignore | report | stopafterpass | stopatonce}

This command determines how a overstretched text formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultpageoverset

`set` fmtfaultpageoverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a page overset formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultpageunderfull

`set` fmtfaultpageunderfull= { ignore | report | stopafterpass | stopatonce}

This command determines how a page underfull formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — (the default) Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaultsoftkeeps

`set` fmtfaultsoftkeeps= { ignore | report | stopafterpass | stopatonce}

This command determines how a soft keeps formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaulttablehorizoverset

`set` fmtfaulttablehorizoverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a table horizontal overset formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtfaulttablevertoverset

`set` fmtfaulttablevertoverset= { ignore | report | stopafterpass | stopatonce}

This command determines how a table vertical overset (in rows) formatting fault is handled while formatting a document instance. The available options are:

- `ignore` — Do not log the fault; do not display the fault in the **Formatter Messages** window.
- `report` — (the default) Log the fault and display the fault in the **Formatter Messages** window.
- `stopafterpass` — Stop formatting after the end of the current formatting pass.
- `stopatonce` — Stop formatting immediately.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtthreshgraphicoverset

`set` fmtthreshgraphicoverset=*size*

This command sets the threshold for determining a graphic overset (horizontal) formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshgraphicoverset=10pt
set fmtthreshgraphicoverset=2pi
set fmtthreshgraphicoverset=7.5mm
set fmtthreshgraphicoverset=1.1cm
set fmtthreshgraphicoverset=0.125in
```

set fmtthreshhdrftroverset

`set` fmtthreshhdrftroverset= *size*

This command sets the threshold for determining a header/footer overset formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmthreshhdrftroverset=10pt
set fmtthreshhdrftroverset=2pi
set fmtthreshhdrftroverset=7.5mm
set fmtthreshhdrftroverset=1.1cm
set fmtthreshhdrftroverset=0.125in
```

set fmtthreshlineoverset

`set` fmtthreshlineoverset=*size*

This command sets the threshold for determining a line overset formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshlineoverset=10pt
set fmtthreshlineoverset=2pi
set fmtthreshlineoverset=7.5mm
set fmtthreshlineoverset=1.1cm
set fmtthreshlineoverset=0.125in
```

set fmtthreshoverstretched

`set` fmtthreshoverstretched= { min | med | max}

This command sets the threshold for determining an overstretched text formatting fault while formatting a document instance. The available options are:

- `min` — Any overstretched text is considered a format fault.
- `med` — (the default) Up to a medium stretch is not considered a fault.
- `max` — No amount of stretch is considered a fault.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtthreshpageoverset

`set` fmtthreshpageoverset=*size*

This command sets the threshold for determining a page overset formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshpageoverset=10pt
set fmtthreshpageoverset=2pi
set fmtthreshpageoverset=7.5mm
set fmtthreshpageoverset=1.1cm
set fmtthreshpageoverset=0.125in
```

set fmtthreshpageunderfull

`set` fmtthreshpageunderfull=*size*

This command sets the threshold for determining a page underfull formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshpageunderfull=10pt
set fmtthreshpageunderfull=2pi
set fmtthreshpageunderfull=7.5mm
set fmtthreshpageunderfull=1.1cm
set fmtthreshpageunderfull=0.125in
```

set fmtthreshsoftkeeps

`set` fmtthreshsoftkeeps= { 1 | 2 | 3 | 4 | 5 | 6}

This command sets the threshold for determining a soft keeps formatting fault while formatting a document instance. The available options are integers from `1` to `6`. The default setting is `1`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

set fmtthreshtablehorizoverset

`set` fmtthreshtablehorizoverset=*size*

This command sets the threshold for determining a table overset (horizontal) formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshtablehorizoverset=10pt
```

```
set fmtthreshtablehorizoverset=2pi
set fmtthreshtablehorizoverset=7.5mm
set fmtthreshtablehorizoverset=1.1cm
set fmtthreshtablehorizoverset=0.125in
```

set fmtthreshtablevertoverset

`set` fmtthreshtablevertoverset=*size*

This command sets the threshold for determining a table overset (vertical, in rows) formatting fault while formatting a document instance. The `size` argument is a non-negative distance and unit specification. Valid units of measurement are points, picas, millimeters, centimeters, and inches. The default setting is `0pt`.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is sent to the Arbortext PE server with a publishing request and applied. After the request is fulfilled, the setting for this option on the server reverts to its last set value.

Examples

```
set fmtthreshtablevertoverset=10pt
set fmtthreshtablevertoverset=2pi
set fmtthreshtablevertoverset=7.5mm
set fmtthreshtablevertoverset=1.1cm
set fmtthreshtablevertoverset=0.125in
```

set fontcoloraqua

`set` fontcoloraqua= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **aqua**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolorblack

`set` fontcolorblack= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **black**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolorblue

`set` fontcolorblue= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **blue**. The value is either a named color or an RGB specification preceded by #.

set fontcolorbrown

`set` fontcolorbrown= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **brown**. The value is either a named color or an RGB specification preceded by #.

set fontcolorgray

`set` fontcolorgray= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray**. For backward compatibility, **gray** is a synonym for **gray3**. The value is either a named color or an RGB specification preceded by #.

set fontcolorgray1

`set` fontcolorgray1= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray1**. The value is either a named color or an RGB specification preceded by #.

set fontcolorgray2

`set` fontcolorgray2= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray2**. The value is either a named color or an RGB specification preceded by #.

set fontcolorgray3

`set` fontcolorgray3= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray3**. The value is either a named color or an RGB specification preceded by #.

set fontcolorgray4

`set` fontcolorgray4= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray4**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolorgray5

`set` fontcolorgray5= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **gray5**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolorgreen

`set` fontcolorgreen= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **green**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolorlime

`set` fontcolorlime= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **lime**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolormaroon

`set` fontcolormaroon= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **maroon**. The value is either a named color or an RGB specification preceded by `#`.

set fontcolornavy

`set` fontcolornavy= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **navy**. The value is either a named color or an RGB specification preceded by #.

set fontcolorolive

`set` fontcolorolive= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **olive**. The value is either a named color or an RGB specification preceded by #.

set fontcolororange

`set` fontcolororange= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **orange**. The value is either a named color or an RGB specification preceded by #.

set fontcolorred

`set` fontcolorred= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **red**. The value is either a named color or an RGB specification preceded by #.

set fontcolorteal

`set` fontcolorteal= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **teal**. The value is either a named color or an RGB specification preceded by #.

set fontcolorviolet

`set` fontcolorviolet= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **violet**. The value is either a named color or an RGB specification preceded by #.

set fontcolorwhite

`set` fontcolorwhite= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **white**. The value is either a named color or an RGB specification preceded by `#`.

set fontcoloryellow

`set` fontcoloryellow= { *colorname* | *#rgbspec*}

This preference determines the color value used to display the font (foreground) color named **yellow**. The value is either a named color or an RGB specification preceded by `#`.

set fontpercent

`set` fontpercent= *n*

This command increases or decreases the point sizes of all fonts in the currently active window to the percentage specified, subject to the availability of fonts. By default, fonts are displayed at 92% of their point size. Fonts cannot be displayed at less than 40% of their point size.

n is a number, excluding a percent (`%`) symbol.

set formatsnapshot

`set` formatsnapshot= { on | off}

When the `set formatsnapshot` command is set to `on`, the contents of the `cache` directory is saved for each formatting pass made on the document. If you report a problem to PTC Technical Support, you may be instructed to run this command and gather the data for analysis. This command works for both local and Arbortext Publishing Engine publishing. The default is `off`.

set formatstatus

`set` formatstatus= { on | off}

When the `set formatstatus` command is set to `on`, Arbortext Editor displays format status messages in the right side of the status bar of the Edit window.

When set to `off`, these messages are disabled. The default value is `on` for all document types except those that have a profiling configuration file.

set formatwarnings

`set` formatwarnings= { on | off}

When this is set to `on`, any warning messages produced when the document is formatted are displayed in a window of their own. `off` is the default.

When this option is set for Arbortext Editor clients using Arbortext Publishing Engine for publishing documents, the setting is applied to a publishing request sent to the Arbortext PE server.

set fosiedit

`set` fosiedit= { on | off}

This command determines whether FOSI specific menus and menu items are displayed in the menu bar. The default is `off`. Before updating menus and menu items for a FOSI, this command calls the `menuloadhook` function on page 962. For example, loading a stylesheet can change custom tables, which in turn can affect the **Insert ► Table** menu items and require the menu to be reloaded. However, simply changing a FOSI that enables or disables specific menu items may not require reloading the menu.

set fosiview

`set` fosiview= { default | edit | html | print}

This command forces a FOSI to be interpreted for a particular view (for example, print) when displayed in any arbitrary view (for example, edit).

For example, with a document open in Arbortext Editor, setting `fosiview` to `print` and updating the generated text causes Arbortext Editor to display the same generated text and formatting used when printing or publishing the document to PDF.

This option affects all views, not just the edit view. For example, setting `fosiview` to `edit` and previewing the document will preview the document using the same generated text and formatting used when editing the document.

Setting `fosiview` to `default` returns Arbortext Editor to interpreting the FOSI according to the current view.

This option can be particularly useful in the following situations:

- When printed generated text is very different from that shown when editing the document.
- When working with PDF bookmarks.
- When working with index formatting.
- When working with table of contents formatting.

set fosiwarnings

`set` fosiwarnings= { on | off}

The `set fosiwarnings` command enables or disables the display of warning messages when a FOSI is attached to format the Editor view. The default is `off`.

set fragmentheader

`set` fragmentheader= { none | comment | document}

This `set` option specifies the format of the fragment header that is written at the beginning of an XML inclusion when it is saved. This header information contains information that is used by Arbortext Editor to allow the inclusion to be edited separately. The default value is `comment`.

- `none` — No header information is written.
- `comment` — The fragment is written as an XML comment.
- `document` — When set to `document`:
  - The fragment is written as a complete XML document.
  - Choosing to create an inclusion from a selection will only be allowed on well-formed selections (selections consisting of a single element).
  - While editing a document that contains expanded XML inclusions, Arbortext Editor will prevent new top-level elements from being added to the expanded inclusions. That is, they will be kept well-formed.
  - The PTC Server connection and repository adapter features that enable you to create an object from a selection in a document require that the selected object be well formed. A well formed object contains the contents of a single element.

set fragmentheaderpreserve

`set` fragmentheaderpreserve= { on | off}

The `set fragmentheaderpreserve` command controls how fragments such as XML inclusions and file entities are written when a document is saved.

- `off` — (The default.) Fragments are written with headers binding each fragment to the document type of the root document. The root document is the document which is ultimately referencing each fragment.
- `on` — Fragments are written with the document type bindings they had when read into memory.

This setting has document scope.

set framesetpath

`set` framesetpath=*d1* [:*dn*]

This command specifies a list of directories to search for framesets. Specify each directory separated by a semicolon. A list of multiple directories must be enclosed in quotation marks.

Framesets must be defined in your `.dcf` file.

If there is an *`Arbortext-path`*`\custom\framesets` subdirectory at startup, the `\custom\framesets` path is automatically prepended to the frameset path. Putting your framesets in the `\custom\framesets` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

You can use the `append_frameset_path` function on page 221 to update the frameset path.

set freeformpis

`set` freeformpis= { on | off}

When an XML instance with no DTD declaration is edited, the user may add new elements and attributes and associate style categories with elements Formerly, for this information to persist across multiple editing sessions required saving the instance in the form of processing instructions (PIs) at the beginning of the file. This is no longer the case.

If `freeformpis` is set to *on* , existing PIs are preserved in Free-form XML documents. If `freeformpis` is set to *off*, existing PIs are removed from Free-form XML documents..

The default value of `freeformpis` is *on* .

set fulljust

`set` fulljust= { on | off}

This command determines whether text can be displayed as fully justified in the Edit window. `off` is the default. The setting does not affect the justification of text inside table cells.

set fullmenus

`set` fullmenus= { on | off}

This command determines whether all available menus and menu items are displayed in the menu bar (if set `on`), or if only the most frequently used items are included (if set `off`). The default is `off`.

set fullname

`set` fullname= *string*

Declares the full name of the current user (as defined by the *user* option) to be *string*. The default value is defined by the system and may be specific to the value of *user*. If this option is not set and the system provides no default, then the value of *user* will be used for *fullname* as well.

This value of *fullname* is added to the document's list of users when the current *user* creates his or her first change record in that document.

Example

```
set fullname="Edgar Allen Poe"
```

set generateuniqueid

`set` generateuniqueid= { on | off}

This option determines whether generated IDs have an additional string appended to them that makes it very likely that the ID will be unique across a set of documents. The additional string is generated by the `generate_id` function and consists of a hyphen followed by eight hexadecimal characters representing the current date and time. Set this option to `on` to append the additional string to generated IDs and to `off` to not append the additional string. The default is `on`. This option has global scope.

set gentext

`set` gentext= { on | off}

This command determines if generated text is displayed in the Edit window document. When set to `on`, the text is displayed, when set to `off` it is not. The default is `on`.

When displayed, the generated text will update according to the setting of the `set gentextautoupdate` on page 816 command. You can also set the generated text display using the **View ► Generated Text ► Show** menu command. The Window preference **Show Full Menus** must be on to access it.

Note

If you are printing large documents in command mode, this option should be turned off. For more information, see Processing speed and generated text.

set gentextautoupdate

`set` gentextautoupdate= { full | partial | none}

This command determines how the generated text in the Edit window updates. It can be changed when generated text is not visible, but the change won't affect program performance until you display the generated text (for example, by setting `gentext` to `on`).

The available settings are:

- `full` — Updates the generated text in the current document, including tables of contents, indices and cross references. This setting works best with small documents without indices.
- `partial` — (default setting) Updates the generated text in the current document, but does not update FOSI Time Independent Variables (TIVs). The most common examples of TIVs are tables of contents, indices and cross references. Other generated text, such as bullets for lists, and numbers for numbered lists or numbered sections, are updated.
- `none` — The generated text does not automatically update. Updates to generated text must be manually initiated by choosing **View ► Generated Text ► Update**. Large documents work best with this setting.

To display generated text, use the `set gentext` on page 815 command. You can set the `gentextautoupdate` option from several places in the user interface. From **View ► Generated Text**, you can choose **Full Auto-Updates**, **Partial Auto-Updates**, or **No Auto-Updates**.

set gentextcurrent

`set` gentextcurrent= { on | off}

When set to `on`, marks generated text for the current view as being up-to-date with regard to changes made in the document.

When Arbortext Editor detects generated text as current, setting this option to `on` does nothing. Arbortext Editor automatically detects whether generated text is current or not. (When a document is changed, generated text is marked as not current. When generated text is updated, it is marked as current.)

You may want to set it to `on` if changes to the environment are made that affect generated text. For example when an ACL variable that is tested by a stylesheet is altered.

You may want to set `gentextcurrent` to `off` in a `formatcontinuehook` function to force generated text to be regenerated before a subsequent formatting pass.

Because this command works on the current view, and document publishing is done in a separate view, `set gentextcurrent` does not affect the generated text for formatted output except in contexts where the print view is the current view. These contexts include the `formatcontinuehook` on page 952 and the `userulehook` on page 969.

This option does not alter whether partial generated text auto-updates are performed.

If you want to force generated text to be regenerated before additional formatting passes are performed, set `gentextcurrent` to `off`.

set gentextdisableautoupdate

`set` gentextdisableautoupdate= { default | small | medium | large | huge}

This command lets you specify when to disable generated text auto-updating to prevent unacceptable processing times for various size documents. The option has document and session scope. Refer to the `set` command on page 744 for more information on specifying document and session scope values.

Generated text automatic updating can slow down processing of documents. The points at which the slowing occurs is based on several factors and varies by document type. The points at which the slowing impacts the user also varies. Setting `gentextdisableautoupdate` to one of the following values can improve document processing times.

- `default` — Sets the document scope value of `gentextdisableautoupdate` to use the session scope value of `gentextdisableautoupdate`. Sets the session scope value to use the system default (`huge`).
- `small` — Generated text automatic updating is disabled on documents with 5,000 or more elements.
- `medium` — Generated text automatic updating is disabled on documents with 20,000 or more elements.
- `large` — Generated text automatic updating is disabled on documents with 50,000 or more elements.
- `huge` — (The default session scope value.) Generated text automatic updating is disabled on documents with 100,000 or more elements.

set gentextfontcolor

`set` gentextfontcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the font color used to display generated text in the Edit window. If `inherit` is set, the generated text is displayed the same as the text of the Edit window document. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

You can determine whether Arbortext Editor ignores this setting by using the `set usecolorsettings` on page 912 option.

Examples

```
set gentextfontcolor=inherit
set gentextfontcolor=red
set gentextfontcolor=#ff0000
```

set gentexttagdisplay

`set` gentexttagdisplay= { none | default | full}

This command determines the display of generated text in a document in the Edit window. The default is `none`.

- `none` — No tags are displayed in generated text.
- `default` — Generated text markup takes on the same setting as the Edit window document tag display.
- `full` — Displays generated text markup, regardless of whether other tags are displayed fully or not.

> **Note**
>
> When the value `full` is set, the tags shown in Editor view will reflect exactly those tags wrapping the generated text in the Generated Text Editor (or the **Generated text** property category) for the element in Arbortext Styler. There are instances in which generated text is not surrounded by any formatting tags by default, for example if it is plain text. In such cases the corresponding generated text in Editor view will also not be surrounded by any tags.

set gentexttrace

`set` gentexttrace= { #NONE | #ALL | [-] *varname1* [, [-] *varname2*] [..., [-] *varnamen*]}

This command traces the listed *varnameN* variables declared in the FOSI's `rsrcdesc`. Use the special variable names `#ALL` or `#NONE` to turn tracing on or off (respectively) for all variables. Multiple variables may be specified by listing names separated by commas. Tracing may be turned off for a variable by preceding the variable name (*varnamen*) with a minus sign.

Note

The command does not control tracing for cross-reference FOSI variables. Use the `set gentextxreftrace` on page 820 command for this purpose.

Examples:

The following command would trace all variables defined in a FOSI's `rsrcdesc`.

```
set gentexttrace=#ALL
```

The following command would trace the variables `divs1ct`, `divs2ct`, and `divs3ct`.

```
set gentexttrace="divs1ct, divs2ct, divs3ct"
```

The following command would trace all variables defined in a FOSI's `rsrcdesc`, except the variables `divs1ct`, `divs2ct`, and `divs3ct`.

```
set gentexttrace=#ALL, "-divs1ct, -divs2ct, -divs3ct"
```

The following command would turn off all FOSI variable tracing.

```
set gentexttrace=#NONE
```

set gentexttracemaxlen

`set` gentexttracemaxlen= *length*

This command sets the length (in characters) of the FOSI variable values (including cross-reference FOSI variable values) displayed in the **FOSI Generated Text Variable Trace** window. If the variable is longer than the specified *length*, "..." is appended to the value. The default (and maximum) length is 4096 characters.

set gentextwarnings

`set` gentextwarnings= { 0 | 1 | 2 | 3}

This command determines what messages Arbortext Editor displays when it detects errors and anomalies in how a FOSI's variables are used. The following settings are available.

- `0` — Display no generated text errors, warnings, or informational messages.
- `1` — Display generated text error messages only.

- `2` — Display generated text error messages and warnings.
- `3` — Display generated text error messages, warnings, and informational messages.

The default setting for Arbortext Editor is `1`; for Arbortext Architect, the default is `3`.

The resulting messages appear in the **FOSI Generated Text Messages** window when formatting is complete, or when an update of the generated text is complete (if triggered independently of formatting).

set gentextxreftrace

`set` gentextxreftrace= { #NONE | #ALL | [-] *varname1* [, [-] *varname2*] [..., [-] *varnamen*]}

This command traces the listed *varnameN* cross-reference FOSI variables. A cross-reference FOSI variable is a variable created using `fillval` to a `savetext:texid` or using `#xref`. Use the special variable names `#ALL` or `#NONE` to turn tracing on or off (respectively) for all cross-reference variables. Multiple cross-reference variables may be specified by listing names separated by commas. Tracing may be turned off for a cross-reference variable by preceding the variable name (*varnamen*) with a minus sign.

Note

The command does not control tracing for other FOSI variables. Use the `gentexttrace` on page 818 option for this purpose.

Examples:

The following command would trace all cross-reference FOSI variables in an instance.

```
set gentextxreftrace=#ALL
```

The following command would trace the cross-reference FOSI variables `intro`, `parttwo`, and `comlist`.

```
set gentextxreftrace=intro, parttwo, comlist
```

The following command would trace all cross-reference FOSI variables in an instance, except the variables `intro`, `parttwo`, and `comlist`.

```
set gentextxreftrace=#ALL, -intro, -parttwo, -comlist
```

The following command would turn off all cross-reference FOSI variable tracing.

```
set gentextxreftrace=#NONE
```

set graphicapptransform

`set` graphicapptransform=*transformstring*

This command specifies a set of graphic file formats that should be transformed to a different graphic file format when using PTC Advanced Print Publisher to publish a document for print or PDF.

The *transformstring* is one or more transformation rules. Multiple rules are separated by a vertical line (`|`). Each transformation rule has two parts separated by a colon (`:`). The first part of the rule is a list of graphic formats to be converted separated by a comma (`,`). The second part of the rule is the graphic file format to which the first set of graphic formats should be transformed during publishing.

The following graphic formats are supported for transformation:

- `sunbm` — Sun Bitmap
- `xbm` — X Bitmap
- `eps` — Encapsulated PostScript file
- `cur`— Windows Cursor file
- `drw` — IslandDraw DRW file
- `sunicon` — Sun Icon file
- `tif` — Tag Image File Format
- `drwunc` — IslandDraw DRAW file (not compressed)
- `png` — Portable Network Graphics
- `wmf` — Windows metafile
- `icon` — Windows icon
- `pcx` — PC Paintbrush File Format
- `bmp` — Windows Bitmap
- `cgm` — Computer Graphics Metafile
- `gif` — Graphic Interchange Format
- `jpg` — Joint Photographic Exports Group
- `calsG4` — CALS raster (G4)
- `svg` — Scalable Vector Graphics
- `idr` — Arbortext IsoDraw file
- `idrz` — Arbortext IsoDraw file
- `iso` — Arbortext IsoDraw file
- `isoz` — Arbortext IsoDraw file
- `edz` — Creo View file

This format is not supported as the target output of a transformation.

- `pvz` — Creo View file

  This format is not supported as the target output of a transformation.

- `pdf` — Portable Document format file

  This format is only supported as the target of a transformation.

- `+` — Represents all graphic file formats that are not web-friendly

  The web-friendly graphic file formats are GIF, JPEG, and PNG.

- `*` — Represents all graphic file formats

For graphic file formats that are also included in the `isoviewfileformats` preference, those formats can be transformed to one of the following types:

- `png`
- `jpg`

All other graphic file formats can be transformed to one of the following types:

- `png`
- `jpg`
- `gif`

  For the graphic file formats specified in the `isoviewfileformats` preference, the target cannot be `gif`.

- `-`

  Indicates that a particular graphic file format should not be transformed. In this case, PTC Advanced Print Publisher will process that format.

Examples:

```
set graphicapptransform=cgm,idr,idrz,iso,isoz:pdf
set graphicapptransform=iso,isoz,pvz:jpg|svg:png
set graphicapptransform=*:png
```

set graphicdefaultwebformat

`set` graphicdefaultwebformat= { png | jpg | gif}

This command specifies the default graphic file format to which graphics that are not web-friendly are converted when publishing for the following types of HTML output:

- **Web**
- **HTML Help**
- **HTML File**

Supported graphic file format values are `png`, `jpg`, or `gif`. The default value is `png`.

set graphicdisplay

`set` graphicdisplay= { on | off}

This command determines whether graphics will be displayed in the Edit window. `on` is the default.

`graphics` is a synonym for `graphicdisplay`.

set graphicfilter

`set` graphicfilter= *filterstring*

This command specifies the default list of graphics file types to display in the **Files of type** pulldown list of the **Locate Graphic File to Reference** dialog box when inserting a new graphic.

filterstring specifies a list of file types separated by a vertical bar "|". There are two parts to each item, which are also separated by a vertical bar "|". The first part is the string to display in the menu or pop-up menu. The second part is the corresponding pattern to use to filter the files displayed in the dialog box. Multiple patterns may be specified by separating them with a semicolon.

The default list contains:

```
All Graphics (*.bmp,*.cgm,*.edz,*.eps,*.gif,*.idr,*.idrz,*.iso,*.isoz,*.jpg,*.pdf,*.png,
*.pvz,*.svg,*.tif)|*.bmp;*.cgm;*.edz;*.eps;*.gif;*.idr;*.idrz;*.iso;*.isoz;*.jpg;*.pdf;*.pn
*.pvz;*.svg;*.tif;*.tiff|
Bitmap Graphics (*.bmp)|*.bmp|
Creo View Graphics (*.edz, *.pvz)|*.edz;*.pvz|
Graphics Interchange Format (*.gif)|*.gif|
IsoDraw Graphics (*.idr, *.idrz, *.iso, *.isoz)|*.idr;*.idrz;*.iso;*.isoz|
JPEG File Interchange Format (*.jpg)|*.jpg|
Portable Network Graphics(*.png)|*.png|
Scalable Vector Graphics (*.svg)|*.svg|
Tag Image File Format (*.tif, *.tiff)|*.tif;*.tiff|
Vector Graphics (*.cgm, *.eps, *.pdf)|*.cgm;*.eps;*.pdf|
All Files|*
```

set graphicrtftransform

`set` graphicrtftransform=*transformstring*

This command specifies a set of graphic file formats that should be transformed to a different graphic file format when using Arbortext Import/Export to export a document to RTF.

The *transformstring* is one or more transformation rules. Multiple rules are separated by a vertical line (`|`). Each transformation rule has two parts separated by a colon (`:`). The first part of the rule is a list of graphic formats to be converted separated by a comma (`,`). The second part of the rule is the graphic file format to which the first set of graphic formats should be transformed during publishing.

The following graphic formats are supported for transformation:

- `sunbm` — Sun Bitmap
- `xbm` — X Bitmap
- `eps` — Encapsulated PostScript file
- `cur`— Windows Cursor file
- `drw` — IslandDraw DRW file
- `sunicon` — Sun Icon file
- `tif` — Tag Image File Format
- `drwunc` — IslandDraw DRAW file (not compressed)
- `png` — Portable Network Graphics
- `wmf` — Windows metafile
- `icon` — Windows icon
- `pcx` — PC Paintbrush File Format
- `bmp` — Windows Bitmap
- `cgm` — Computer Graphics Metafile
- `gif` — Graphic Interchange Format
- `jpg` — Joint Photographic Exports Group
- `calsG4` — CALS raster (G4)
- `svg` — Scalable Vector Graphics
- `idr` — Arbortext IsoDraw file
- `idrz` — Arbortext IsoDraw file
- `iso` — Arbortext IsoDraw file
- `isoz` — Arbortext IsoDraw file
- `edz` — Creo View file
- `pvz` — Creo View file
- `+` — Represents all graphic file formats that are not web-friendly

  The web-friendly graphic file formats are GIF, JPEG, and PNG.
- `*` — Represents all graphic file formats

If the source graphic file format is included in the `isoviewfileformats` preference, those graphic file formats can be transformed to one of the following types:

- `png`
- `jpg`
- `cgm`
- `eps`

All other graphic file formats can be transformed to one of the following types:

- `png`
- `jpg`
- `gif`
- `-`

  Indicates that a particular graphic file format should not be transformed.

Examples:

```
set graphicrtftransform=iso,isoz,edz,pvz:cgm|svg:png
set graphicrtftransform=*:png
```

set graphicwebtransform

`set` graphicwebtransform=*transformstring*

This command specifies a set of graphic formats that should be transformed to a different graphic format when publishing for the following types of HTML output:

- **Web**
- **HTML Help**
- **HTML File**

The *transformstring* is one or more transformation rules. Multiple rules are separated by a vertical line (`|`). Each transformation rule has two parts separated by a colon (`:`). The first part of the rule is a list of graphic formats to be converted separated by a comma (`,`). The second part of the rule is the graphic file format to which the first set of graphic formats should be transformed during publishing.

The following graphic formats are supported for transformation:

- `sunbm` — Sun Bitmap
- `xbm` — X Bitmap
- `eps` — Encapsulated PostScript file
- `cur`— Windows Cursor file

- `drw` — IslandDraw DRW file
- `sunicon` — Sun Icon file
- `tif` — Tag Image File Format
- `drwunc` — IslandDraw DRAW file (not compressed)
- `png` — Portable Network Graphics
- `wmf` — Windows metafile
- `icon` — Windows icon
- `pcx` — PC Paintbrush File Format
- `bmp` — Windows Bitmap
- `cgm` — Computer Graphics Metafile
- `gif` — Graphic Interchange Format
- `jpg` — Joint Photographic Exports Group
- `calsG4` — CALS raster (G4)
- `svg` — Scalable Vector Graphics
- `idr` —Arbortext IsoDraw file
- `idrz` — Arbortext IsoDraw file
- `iso` — Arbortext IsoDraw file
- `isoz` — Arbortext IsoDraw file
- `edz` — Creo View file
- `pvz` — Creo View file
- `+` — Represents all graphic file formats that are not web-friendly

  The web-friendly graphic file formats are GIF, JPEG, and PNG.
- `*` — Represents all graphic file formats

These graphic file formats can be transformed to one of the following types:

- `png`
- `jpg`
- `gif`

  For the graphic file formats specified in the `isoviewfileformats` preference, the target cannot be `gif`.
- `webcgm`

  Indicates that a particular graphic file format should be transformed to CGM. In this case, the graphics can be viewed in HTML output in any viewer that supports WebCGM. You can use this setting in combination with the `set`

`cgmprofile` preference to indicate a particular CGM profile to use for the converted graphics.

For example, to transform CGM and Arbortext IsoDraw graphics to WebCGM 1.0, you would set these preferences to the following values:

```
set graphicwebtransform=iso,isoz,cgm:webcgm
set cgmprofile=WebCGM_1.0
```

If `cgmprofile` is not set, then the graphics are transformed to the WebCGM 2.0 profile.

- `iview`

  Indicates that a particular graphic file format should not be transformed, and that in the HTML output graphics of that type will be displayed by using Arbortext IsoView.

- `-`

  Indicates that a particular graphic file format should not be transformed. In this case, if a specified graphic file format is not web-friendly, graphics of that type can only be displayed in a web browser if the stylesheet provides necessary HTML markup. For example, if the `graphicwebtransform` is `svg:-`, the stylesheet needs to produce HTML markup that displays SVG graphics.

When determining whether to transform a graphic, Arbortext Editor checks several ACL preferences. The `graphicwebtransform` preference takes precedence over all other preferences, except `cgmprofile`. The following other preferences can affect graphics transformations:

- `intelligentgraphicsconversion`
- `graphicdefaultwebformat`
- `isoviewfileformats`
- `cgmprofile`

For example, if `graphicwebtransform` is set to `iso,pvz:iview`, but `isoviewfileformats` does not include `pvz`, that graphic file format will still be displayed by Arbortext IsoView in the HTML output.

The **graphicwebtransform** preferences also overrides the value of the **Convert Intelligent Graphics** check box in the publishing dialog boxes.

For each graphic during publishing, Arbortext Editor checks to see whether that graphic file format is in one of the first part of the `graphicwebtransform` setting. If it is, Arbortext Editor transforms the graphic based on the target of the rule. If the format is not included in `graphicwebtransform`, Arbortext Editor processes the graphic based on whether the graphic is intelligent and whether the `intelligentgraphicsconversion` preference is set to `on`. If `intelligentgraphicsconversion` is set to `off` and the graphic is an intelligent graphic, the graphic is not transformed. Otherwise, if the graphic is not

web-friendly, Arbortext Editor will transform the graphic based on the graphic file format specified by the `graphicdefaultwebformat` preference. A graphic is considered an intelligent graphic if it is a format specified by the `isoviewfileformats` preference.

Examples:

```
set graphicwebtransform=iso,isoz,edz,pvz:cgm|svg:png
set graphicwebtransform=*:png
```

set graphicspath

`set` graphicspath=*d1* [:*dn*]

This command specifies a list of directories to search for graphic files when their references do not specify an absolute path name (each directory separated by a semicolon). The initial setting is the value of the *APTGRPATH* environment variable if set. Otherwise, it defaults to the `graphics` subdirectory of *`Arbortext-path`*. If a `graphics` subdirectory exists in either the *`Arbortext-path`*`\application` or *`Arbortext-path`*`\custom` directory, that subdirectory is also part of the default graphics path. Use `%D` to refer to the *`Arbortext-path`*`\graphics` directory. Use `%B` to refer to directories relative to the current document's directory.

Any base URI and the document's parent directory (if different from the base URI) are included by default in the directories searched for graphics. When any `set graphicspath` directory parameter is set to a value of `%^`, the parent document directory will **not** be included when searching for graphics. Only the paths specified will be searched. If you want to include the current working directory in the graphics path, add a period (`.`) to the list of directories. However, it is recommended that you do not include the current working directory in your graphics path list, as this can cause inconsistent results.

If there is an *`Arbortext-path`*`\custom\graphics` subdirectory at startup, the `\custom\graphics` path is automatically prepended to the graphics path. Putting your graphics in the `\custom\graphics` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

The *`Arbortext-path`*`\custom\graphics` directory is also automatically included if you specify `%D` in the path.

Example

```
set graphicspath="c:\\Program Files\\Arbortext\\editor\\graphics;%^;
 c:\\Arbortext\\editor\\packages\\graphics"
```

A list of multiple directories must be enclosed in quotation marks.

You can use the `append_graphics_path` function on page 221 to update the graphics path.

set helpfontpercent

`set` helpfontpercent= *n*

This command increases or decreases the point sizes of all fonts in all message windows to the percentage specified, subject to the availability of fonts. The default value of *n* is system dependent. Fonts cannot be displayed at less than 40% of their point size. This command does not affect the font size of Help Center content.

n is a number, excluding a percent (`%`) symbol.

set hiddentagscan

`set` hiddentagscan= { on | off}

Controls whether elements configured by the `tag_display` command on page 719 to not be displayed will be recognized and operated on by find and spelling check operations. When `set hiddentagscan` is set to `on`, the content of elements that are not displayed will be operated on. The default setting is `off`, meaning the non-displayed content will not be operated on.

set hidesuppressed

`set` hidesuppressed= { on | off}

When `set hidesuppressed` is set to `on`, the content of elements that are configured by a Arbortext Styler stylesheet to be hidden or by a screen FOSI to be suppressed is hidden in the Editor view. The default setting is `off`, meaning suppressed content is displayed in the Editor view. This allows you to edit content that is suppressed where it occurs in the document, but that is inserted in generated text.

For example, you might add end note content to a paragraph, and then create generated text that inserts the content of the end note. When `set hidesuppressed` is `on`, the end note content in the paragraph is hidden in Editor view, while the generated text version displays. When set to `off`, the content displays in both the paragraph and the generated text version.

If you insert a suppressed element within generated text, the content of the element is not shown in the Editor view regardless of the setting of this option.

set highlightinvalidmarkup

`set` highlightinvalidmarkup= { on | off}

This command modifies the display of invalid markup. By default, invalid markup is shown in red strike-through. If this command is set to `off`, invalid markup is displayed in the same manner as valid markup.

The default is `on`.

set htmlextension

`set` htmlextension= *ext*

This command specifies the extension to use for newly saved HTML files. `htm` is the default. The **Save As** dialog box and `save_as` command add this extension if the specified file does not already have an extension and the `htmlextension` option is non-null.

Do not include a period as part of the extension.

set htmlhelpstylesheet

`set` htmlhelpstylesheet= { *name* | none}

This command specifies the stylesheet to be used as the default for the **Publish ► For HTML Help** option on the **File** menu.

name is the path and file name of a `.style` or `.xsl` stylesheet file. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, and then a `.xsl` file.

The stylesheet ID processing instruction must specify `htmlhelp` as the publishing type.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`. The Arbortext PE server must have the HTML Help Workshop installed to create the `.chm` file.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the

server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:

```
set htmlhelpstylesheet=D:\ArbortextUser\axdocbook.xsl
```

- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.

  ```
  set htmlhelpstylesheet=axdocbook.xsl
  ```

If `htmlhelpstylesheet` is set to `none` or is not set, Arbortext Editor and Arbortext Publishing Engine use the Editor/Default stylesheet.

Examples

```
set htmlhelpstylesheet=memo2
set htmlhelpstylesheet=~/mydocs/memo/memo2.style
set htmlhelpstylesheet=http://www.some_site.com/examples/techman_hh.xsl
```

set htmlstylesheet

`set` htmlstylesheet= { *name* | none}

This command specifies the stylesheet to be used for the **Save as HTML** and **Publish ► HTML File** options on the **File** menu and for browser display.

name is the path and file name of a FOSI, XSL, or Arbortext Styler stylesheet. If you do not specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, then a `.fos` file, and finally a `.xsl` file.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:

```
set htmlstylesheet=D:\ArbortextUser\axdocbook.xsl
```

- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.
  ```
  set htmlstylesheet=axdocbook.xsl
  ```

If `htmlstylesheet` is set to `none` or is not set, Arbortext Editor and Arbortext Publishing Engine use the Editor/Default stylesheet.

Note

The **Save as HTML** option only uses FOSI stylesheets. So if *name* specifies an XSL or Arbortext Styler stylesheet, **Save as HTML** uses the stylesheet specified for print published output. If the print published output stylesheet is an XSL or Arbortext Styler stylesheet, **Save as HTML** uses the Editor stylesheet.

Examples

```
set htmlstylesheet=memo2
set htmlstylesheet=~/mydocs/memo/memo2.fos
set htmlstylesheet=http://www.some_site.com/examples/techman.xsl
```

set hyperlinkmenus

`set` hyperlinkmenus= { on | off}

When this command is set to `on`, the **Create Hyperlink**, **Delete Hyperlink**, and **Show Hyperlinks** commands are added to the **Tools** menu. When set to `off`, these commands are not available on the **Tools** menu.

The default is `off` because these menu commands have been replaced by **Insert ▸ Link** and **Insert ▸ Link Target**. Set this command to `on` if your document type does not support link markup.

set importexportpath

`set` importexportpath=*dir1* [:*dirn*]

This command specifies the directory that Arbortext Import/Export searches for library, configuration, MapTemplate, and RTF style template files supporting Arbortext Import/Export Import MapTemplate and Export stylesheet development.

set includefontcolor

`set` includefontcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the font color used to display XML inclusions in the Edit window. If `inherit` is set, the included text is displayed the same as the text of the Edit window document. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

You can determine whether Arbortext Editor ignores this setting by using the `set usecolorsettings` on page 912 option.

Examples

```
set includefontcolor=inherit
set includefontcolor=red
set includefontcolor=#ff0000
```

set indent

`set` indent= { on | off}

When this command is set to `on`, divisions are inset from the left margin of the Edit window display according to level (for example, sections and section headings are indented more than chapters). When set to `off`, divisions are not indented.

This command does not affect the indentation of the formatted document.

set inlineapplicabilitycolor

`set` inlineapplicabilitycolor=*color*

Changes the current color setting for applicability highlighting.

- *color* — the required color, for example:
  ```
  set inlineapplicabilitycolor=blue
  ```

The default color is `lime`.

You can also set this option with the `inlineapplicabilityfontcolor` advanced preference.

Applicability shading is activated with the `set showprofileshading` option.

set inlineapplicabilitynamecheck

`set` inlineapplicabilitynamecheck= { on | off}

Specifies whether to allow values in inline applicability expressions that are not defined in the referenced option sets.

If this option is activated with `set inlineapplicabilitynamecheck=on`, expressions that use undefined values are not allowed. For example, if a product's defined options are `cat` and `mouse`, the inline applicability expression "`cat`" `or` "`dog`" will not be allowed when the expression is validated in the **Apply Inline Applicability** dialog box.

You can also set this option with the `inlineapplicabilitynamecheck` advanced preference.

set inlineapplicabilitysyntax

`set` inlineapplicabilitysyntax=*syntax*

Sets the applicability syntax for the current Arbortext Editor environment.

- *syntax* — the required applicability syntax, for example:
  `set inlineapplicabilitysyntax=ATO`

Setting this option with any non-blank value also enables Arbortext Editor's interface for authoring inline applicability.

You can also set this option with the `inlineapplicabilitysyntax` advanced preference.

Arbortext Editor ships with two applicability syntaxes: APEX and ATO.

To register a custom syntax, use the `registerApplicabilitySyntax()` applicability function.

To retrieve the syntax that is currently specified for the environment, use `eval option(inlineapplicabilitysyntax).`

set inlineapplicabilityui

`set` inlinepplicabilityui= [symbols | words | wordsexclude]

This command specifies the way in which the constructs in inline applicability expressions are represented when in Arbortext Editor's inline applicability interface.

- *symbols* — use symbols

  Not: !

  And: &&

  Or: ||
- *words* — use words, variety 1 (default)

Not: NOT

And: AND

Or: OR

- *wordsexclude* — use words, variety 2

  Not: EXC

  And: AND

  Or: OR

For example:
```
set inlineapplicabilityui=symbols
```

Note

When an inline applicability expression is assigned to an element, the format of the constructs is converted (if necessary) to match the applicability syntax that is active for the current environment.

You can also set this option with the `inlineapplicabilityui` advanced preference.

set inlineediting

`set` inlineediting= { on | off}

This option turns inline editing of file entities and XML inclusions on and off. The default is `on`. This setting does not apply to documents, entities, and inclusions stored CMS and accessed using a repository adapter.

set inputmode

`set` inputmode= { insert | overstrike | toggle}

This command determines whether new text can be inserted at the point of the cursor (the default) or will overwrite existing text. `toggle` changes from the existing setting to the other setting.

set insertpreviewlinktext

`set` insertpreviewlinktext= { on | off}

This option determines whether preview text from the link target is inserted and displayed in the Arbortext Editor window for DITA `xref` and `link` tags. The preview text is taken from the content of either the `navtitle`, `title`, or `searchtitle` tags and from the `shortdesc` tag in the link target. The preview text is highlighted in the Arbortext Editor window.

Set this option to `on` to insert the preview text and to `off` to not insert it. The default is `on`.

set insertsymboldlgnosymbols

`set` insertsymboldlgnosymbols= { on | off}

If set to `on` (the default), the **Symbol** tab of the **Insert Symbol** dialog box is disabled, allowing the user to insert only character entities from the **Character Entities** tab.

If `insertsymboldlgnosymbols` is set to `off` (the default), the **Symbols** tab on the **Insert Symbols** dialog box is enabled.

set insertsymbolfontpi

`set` insertsymbolfontpi= { on | off}

If set to `on` (the default), when inserting symbols from the **Symbol** tab of the `Insert Symbol` dialog box, a `_font` PI (processing instruction) is inserted around the symbol character code if the inserted symbol is of a different font family than the font family at the current cursor location. The PI will be added if Arbortext Editor is unable to translate the requested character from the Symbol character set to Unicode.

If `insertsymbolfontpi` is set to `off`, the Unicode character value is inserted with no `_font` PI.

set intelligentgraphicsconversion

`set` intelligentgraphicsconversion= { on | off | prompt}

This command specifies whether converting intelligent graphics to web-friendly graphic formats is needed for the **File ► Publish ► For Web** and **File ► Publish ► HTML File** operations.

If set to `on`, all intelligent graphics in the document will be converted to web-friendly graphics in the published HTML file. EDZ graphics will be converted to GIF. ISO/IDR graphics will be converted to PNG or JPEG, depending on the setting of the `graphicdefaultwebformat` option. If set to `off`, intelligent

graphics will be included in the published HTML file. If set to `prompt`, the **Convert Intelligent Graphics** check box is enabled in the **Publish for Web** and **Publish HTML File** dialog boxes.

The default is `prompt`.

set isoviewdownloaduri

`set` isoviewdownloaduri=*uri*

This command specifies a *uri* where the Arbortext IsoView web installer is located. The *uri* parameter contains a URI (Uniform Resource Identifier) value.

Arbortext Editor writes the specified *uri* location to a generated HTML file when an intelligent graphic is published in the HTML file. This enables Internet Explorer to install Arbortext IsoView when the HTML file is displayed in the browser. Arbortext IsoView enables users to interact with the intelligent graphics in the HTML file.

If `isoviewdownloaduri` is not set or contains an empty value, you can still view intelligent graphics in a published HTML file. To support viewing intelligent graphics, Arbortext IsoView must already be installed on the workstation where Internet Explorer is running.

Refer to the Arbortext IsoView documentation for more information about Arbortext IsoView.

Examples:

```
set isoviewdownloaduri=http://www.acme.com/isoview/isoviewx6_PTC.cab
```

set isovieweditorfileformats

`set` isovieweditorfileformats=" [cgm] [;idr] [;idrz] [;iso] [;isoz] [;svg] "

By default, Arbortext Editor uses different underlying technologies to display images of different formats. The `set isovieweditorfileformats` option lets you override these default settings to specify additional graphic file formats display using the embedded Arbortext IsoView control in Arbortext Editor.

This preference has no default value in the full installation of Arbortext Editor, where the `set isoviewfileformats` on page 838 preference specifies both the graphic file formats displayed and published using Arbortext IsoView.

The arguments are the file extensions for the graphic types separated by semicolons. If you set the value to more than one graphic type, enclose the arguments in quotes.

Refer to the Arbortext IsoView documentation for more information about Arbortext IsoView. .

Examples:

```
set isovieweditorfileformats=cgm
set isovieweditorfileformats="svg;cgm"
```

set isoviewfileformats

`set` isoviewfileformats=" [cgm] [;idr] [;idrz] [;iso] [;isoz] [;svg] "

This command specifies both the graphic file formats that should be displayed using the embedded Arbortext IsoView control in Arbortext Editor and the graphic types that should be published using Arbortext IsoView. By default, the following types of graphics are displayed in the embedded dialog box:

- CGM (`.cgm`)
- IDR (`.idr`)
- IDRZ (`.idrz`)
- ISO (`.iso`)
- ISOZ (`.isoz`)

You can set the dialog box to also display SVG (`.svg`) graphics. The arguments are the file extensions for the graphic types separated by semicolons. If you set the value to more than one graphic type, you must enclose the arguments in quotes. If you set the value of this option to an empty string, the Arbortext IsoView control will not be used to display any graphics.

Publishing intelligent graphics using Arbortext IsoView is also supported for all intelligent graphics types. Refer to the Arbortext IsoView documentation for more information about Arbortext IsoView.

Examples:

```
set isoviewfileformats=iso
set isoviewfileformats="iso;svg"
```

set isoviewhighlightcolor

`set` isoviewhighlightcolor= { *colorname* | *#rgbspec*}

This command determines the color used to highlight an object in an intelligent graphic displayed using the embedded Arbortext IsoView control in Arbortext Editor. The highlight color is used when the object is the target of a link. The default value is `red`. The supported values are either one of the Arbortext supported named colors or an RGB specification value. Refer to the glossary in the Arbortext Editor help for more information about named colors and RGB values.

You can specify the highlighting style with the `isoviewhighlightstyle` command.

The value of this preference is used in both editing and publishing. The value is passed along to HTML output during publishing.

Examples:

```
set isoviewhighlightcolor=green
set isoviewhighlightcolor=#33FFFF
```

set isoviewhighlightstyle

`set` isoviewhighlightstyle= { circle | fill | flash | frame}

This command determines the style used to highlight an object in an intelligent graphic displayed using the embedded Arbortext IsoView control in Arbortext Editor. The highlight style is used when the object is the target of a link. The following values are supported:

- **circle** — a circle is drawn around the object
- **flash** — the object flashes
- **fill** — the object's color is changed
- **frame** — a thick line is drawn along the outline of the object

  This is the default value. Note that 3D intelligent graphics do not support this style. When the highlight style is set to **frame**, 3D intelligent graphics are highlighted with the **fill** style.

You can specify the highlighting color by using the `isoviewhighlightcolor` preference. The highlighting color is used by all four styles.

The value of this preference is used in both editing and publishing. The value is passed along to HTML output during publishing.

Example:

```
set isoviewhighlightstyle=flash
```

set javaclasspath

`set` javaclasspath=*dir1* [:*dirn*]

This command specifies the list of directories that the Arbortext Editor embedded Java Virtual Machine (JVM) should search to locate Java classes when Java methods are called from within Arbortext Editor. The initial path setting of `set javaclasspath` is empty.

Whether `javaclasspath` is set or not, Arbortext Editor automatically includes the distributed Java JAR files in the directory *`Arbortext-path`*`\lib\classes`.

Note

If there are classes in the *Arbortext-path*\custom\classes subdirectory at startup, the \custom\classes path is automatically prepended to the class path. If there are any subdirectories of the \classes directory that contain classes, those subdirectories are also prepended to the path.

The setting for javaclasspath is only evaluated when the first java_*type* function is called in an Arbortext Editor session. Subsequent changes to set javaclasspath will not affect the running Java Virtual Machine unless you exit Arbortext Editor and start a new session. Consequently, when using the java_*type* functions, you should set javaclasspath before invoking a java_*type* function.

Examples:

```
set javaclasspath="c:\\winnt\\java\\classes;c:\\myjava\\myjava.jar"
```

Multiple directories are delimited with semicolons and the entire set enclosed in quotes.

If there is an *Arbortext-path*\custom\classes subdirectory at startup containing any JAR files (.jar), the path for each .jar file is automatically prepended to the Java class path for Arbortext Editor. Then the \custom\classes path is prepended, which automatically includes any compiled Java .class files in it. Putting your .class and .jar files in the \custom\classes subdirectory makes them automatically available, avoiding manual steps to add them to the path.

You should use the append_javaclass_path function on page 221 to update the Java class path for Arbortext Editor to avoid problems resulting from overlooking a Java class path when using set javaclasspath.

If set javaclasspath is used with Arbortext Publishing Engine, you should protect it from being sourced again after the JVM has started. Put the set javaclasspath statement inside an appropriate if statement. For example:

```
# Try to change this value only if the JVM has not started yet.
# This protects from failure in case this file is sourced again
# after the JVM has already started.
if (!java_init()) {
 set javaclasspath="c:\\winnt\\java\\classes;D:\\myjava\\myjava.jar"
}
```

set javadebugport

set javadebugport= { off | auto | *n*}

When Arbortext Editor loads the Java Virtual Machine (JVM), it checks the value of the `set javadebugport` option. If `set javadebugport` is set to a positive number, Arbortext Editor turns on the JVM debug mode and establishes communication with Java debuggers by using the socket port number specified by `set javadebugport` . If `set javadebugport` is set to `auto`, Arbortext Editor randomly selects an unused socket port number, sets `set javadebugport` to the selected port number, and performs the remaining tasks for establishing communication. If the `set javadebugport` is set to `off` (the default), Arbortext Editor will not turn on the JVM debug mode.

Caution

If you set `set javadebugport` to `on` or `auto`, you must include the Java Development Kit (JDK) `bin` directory in your *PATH* environment variable. If you do not add the JDK `bin` directory to your *PATH* environment variable, Arbortext Editor will terminate when it invokes the JVM.

To establish communication between the Java debugger and Arbortext Editor, pass the socket port number to the Java debugger. For example, if the `set javadebugport` is set to 3539 and the `jdb` (Java debugger) is running on the same machine as Arbortext Editor, issue the following command:

```
c:\> jdb -connect com.sun.jdi.SocketAttach:port=3539
```

Refer to OracleJava documentation on *`jdb` — The Java debugger* for information on the `jdb` command and its options, which is available from the java.sun.com web site.

The setting for `javadebugport` is only evaluated when the first `java_type` function is called in a Arbortext Editor session. Subsequent changes to `set javadebugport` will not affect the running Java Virtual Machine unless you exit Arbortext Editor and start a new session. Consequently, when using the `java_type` functions, you should set `javadebugport` before invoking a `java_type` function.

Example:

```
set javadebugport=auto
# Load the JVM
java_console()
# Get the socket port number for Java debugging
eval option('javadebugport')
```

If `set javadebugport` is used with Arbortext Publishing Engine, you should protect it from being sourced again after the JVM has started. Put the `set javadebugport` statement inside an appropriate `if` statement. For example:

```
# Try to change this value only if the JVM has not started yet.
# This protects from failure in case this file is sourced again
# after the JVM has already started.
if (!java_init()) {
```

```
 set javadebugport=8090
}
```

set javascriptinterpreter

`set` javascriptinterpreter= { rhino | jscript}

The `javascriptinterpreter` option specifies the default JavaScript engine to be used when a file's extension is the only determining factor as to which engine to use.

If `javascriptinterpreter` is set to `rhino`, files with a `.js` extension will be processed by the Rhino JavaScript engine. If `javascriptinterpreter` is set to `jscript`, files with a `.js` extension will be processed by the Microsoft JScript engine.

The default value of `javascriptinterpreter` is `rhino`.

set javavmargs

`set` javavmargs=*vmargs*

The value of this option will be passed to the Java Virtual Machine (JVM) as command line arguments when the JVM is loaded. This option can't specify the class path. Set the class path using the `set javaclasspath` option.

The setting for `javavmargs` is only evaluated when the first `java_`*`type`* function is called in a Arbortext Editor session. Subsequent changes to `set javavmargs` will not affect the running Java Virtual Machine unless you exit Arbortext Editor and start a new session. Consequently, when using the `java_`*`type`* functions, you should set `javavmargs` before invoking a `java_`*`type`* function.

Example:

```
# Define system property my.preferences and set the JVM
# initial memory pool to six megabytes.
set javavmargs="-Dmy.preferences=c:\prefs.txt -Xms6m"
```

If `set javavmargs` is used with Arbortext Publishing Engine, you should protect it from being sourced again after the JVM has started. Put the `set javavmargs` statement inside an appropriate `if` statement. For example:

```
# Try to change this value only if the JVM has not started yet.
# This protects from failure in case this file is sourced again
# after the JVM has already started.
if (!java_init()) {
 set javavmargs="-Xms10m"
}
```

When using XSL-FO to publish documents of any size, XSL-FO may require more Java stack size than that set by default. If the stack size is insufficient, Arbortext Publishing Engine or Arbortext Editor may terminate unexpectedly. If this scenario occurs, increase the Java stack size by adding the following line to an ACL file stored in the `custom\init` directory:

```
set javavmargs="-Xss2m"
```

set javavmmemory

`set` javavmmemory= *n*

This option specifies the maximum size, in megabytes, of the Java Virtual Machine (JVM) memory allocation pool. The default value is 2048 for 64-bit machines, and 256 for 32-bit machines. Reduce this value if Arbortext Editor or Arbortext Publishing Engine cannot allocate enough memory as specified by this option.

Note

If the environment variable APTJAVAVMMEMORY is set, the required size of the JVM is set at startup of Arbortext Editor, and any subsequent `set javavmmemory` commands are ignored.

If APTJAVAVMMEMORY is not set, the setting for `set javavmmemory` is evaluated when the first `java_`*`type`* function is called in a Arbortext Editor session. Subsequent changes to `set javavmmemory` will not affect the running Java Virtual Machine unless you exit Arbortext Editor and start a new session. Consequently, when using the `java_`*`type`* functions, you should set `javavmmemory` before invoking a `java_`*`type`* function.

If `set javavmmemory` is used with Arbortext Publishing Engine, you should protect it from being sourced again after the JVM has started. Put the `set javavmmemory` statement inside an appropriate `if` statement. For example:

```
# Try to change this value only if the JVM has not started yet.
# This protects from failure in case this file is sourced again
# after the JVM has already started.
if (!java_init()) {
 set javavmmemory=500
}
```

If you experience difficulty starting the JVM, it's possible that the memory allocation is higher than your operating system can accommodate. Try setting the `javavmmemory` value lower (for example, on a 32-bit machine, to a value below 900) to see if the JVM will start.

On a 64-bit machine, the maximum size is essentially unlimited. However, setting the value too high can decrease performance due to paging of virtual memory to disk. Commonly, it should be substantially less than the physical RAM on the system. For example, if you have 4GB RAM, try setting `javavmmemory` to `2000`. If publishing extremely large documents results in failures with the error `FATAL ERROR - Java has run out of memory.` messages, try increasing the value until the error no longer occurs.

For more information about tuning the JVM, go to:

java.sun.com/performance/reference/whitepapers/tuning.html

set javavmpath

`set` javavmpath= { *vmpath*}

This command specifies the type and location of the Java Virtual Machine (JVM) to use when calling Java classes from Arbortext Editor.

This command can be set to either a full path (*vmpath*).

Note

The environment variable *APTJAVAVMPATH* specifies the path of the Java Runtime Environment installation. If this environment variable is set, it takes precedence over a set `javavmpath` command.

If the option is set to a full path (*vmpath*), then Arbortext Editor uses the specified JVM for the current session.

If this option is not set, Arbortext Editor checks the Windows registry to locate an Oracle JVM. If this check is not successful, then Arbortext Editor will have no JVM available and will return an error if you attempt to invoke Java code from within Arbortext Editor.

The setting for `javavmpath` is only evaluated when the first `java_type` function is called in an Arbortext Editor session. Subsequent changes to `set javavmpath` will not affect the running Java Virtual Machine unless you exit Arbortext Editor and start a new session. Consequently, when using the `java_type` functions, you should set `javavmpath` before invoking a `java_type` function.

Examples:

```
set javavmpath=msjava.dll
```

If `set javavmpath` is used with Arbortext Publishing Engine, you should protect it from being sourced again after the JVM has started. Put the `set javavmpath` statement inside an appropriate `if` statement. For example:

```
# Try to change this value only if the JVM has not started yet.
```

```
# This protects from failure in case this file is sourced again
# after the JVM has already started.
if (!java_init()) {
 set javavmpath="D:\jre\lib\jvm.dll"
}
```

set keymap

`set` keymap= { user | system | *name*}

This command attaches the specified keymap to the current window. *name* is the name of a keymap previously created by the `define_keymap` or `copy_keymap` commands, or is a keymap automatically created by a `map` command on a window with no attached keymap. In the latter case, the name of the keymap is the same as the window name, that is, as returned by `window_name()`.

Any subsequent `map` commands in the current window not specifying a keymap argument will affect this keymap.

If the keymap is `user`, then the user keymap for the current window class is used. For example, the edit class map can be used for an Edit window. If the keymap is `system`, then the system level keymap for the current window class is used. The system map does not contain any user-level mappings. If a `map` command is issued with the `system` keymap attached, then `map` creates a new keymap with a name the same as the window name. This then becomes the current keymap. This automatically created keymap is deleted when the window is destroyed, if no other window has attached it.

If *name* specifies a prefix keymap, then the keymap remains attached only for the next key or button event. After the key or button event is looked up in the keymap, the keymap for the active window reverts to the previous keymap, unless the mapping for the event switches keymaps explicitly with another `set keymap` command. If the prefix keymap does not contain a mapping for the next key or button event, an error is signalled and the keymap reverts to the previous setting.

Examples

```
set keymap=user
define_keymap -prefix ctrl_x_map
define_keymap -prefix ctrl_x_4_map
map @ctrl_x_4_map d directory
map @ctrl_x_map 4 set keymap=ctrl_x_4_map
map control-x set keymap=ctrl_x_map
```

set language

`set` language= *string*

This command enables you to switch the Proximity/Merriam-Webster Linguibase to a different language dictionary. The default dictionary is English (US).

Dictionaries are available for the following languages:

- Arabic
- Bulgarian
- Brazilian
- British
- Canadian-English
- Canadian-French
- Catalan
- Croatian
- Czech
- Danish
- Dutch
- English
- Estonian
- Finnish
- French
- German
- Greek
- Hebrew
- Hungarian
- Italian
- Latvian
- Legal+English
- Lithuanian
- Medical+English
- Norwegian
- Nynorsk
- Polish
- Portuguese
- Romanian
- Russian
- Slovak
- Slovenian

- Spanish
- Swedish
- Swiss-German
- Thai
- Turkish
- Vietnamese

Note

Norwegian is Bokmal. Medical+English is an English-based medical spell checking dictionary. Legal+English is an English-based legal spell checking dictionary.

To set a different default language to be used for a document, add the `set language` command to the document's `.acl` file. You can retrieve the list of language dictionaries being used by calling the `languages` on page 398 function.

set libpath

`set` libpath=*dir1* [:*dirn*]

This command specifies a search path for shared modules, formatting files, and font configuration files. The default path is *`Arbortext-path`*`\lib`. If you are running in a supported locale, the default path also includes the appropriate subdirectory of *`Arbortext-path`*`\lib\locale`.

Each directory specified must be separated by a semicolon. You can specify the `%D`, `%H`, or `%L` symbolic parameters in the path list to include the default search path.

If an *`Arbortext-path`*`\custom\lib` subdirectory exists at startup, the `custom\lib` path is automatically prepended to the path. If you are running in a supported locale, the default path also includes the associated locale subdirectory of *`Arbortext-path`*`\lib\locale`.

Putting your custom shared modules, formatting files, and font configuration files in the `\custom\lib` directory or one of its *`Arbortext-path`*`\lib\locale` subdirectories makes them automatically available, avoiding manual steps to add them to the path.

Example

```
set libpath="c:\mylib;%D"
```

set liteui

`set` liteui= { on | off}

This command determines the user interface used by Arbortext Editor. If set to `on`, this command enables a simplified user interface with a reduced set of menus and a single tool bar. If set to `off`, the standard user interface is presented. The default is `off`.

set loadmessages

`set` loadmessages= { on | off}

The `set loadmessages` command enables or disables the display of "Loading file" messages in the message area at the bottom of the Edit window when a command file is read. The default is `off`.

set loadpath

`set` loadpath=*d1* [:*dn*]

The `loadpath` option lets you specify the list of directories to search when loading a *package* given in the `require` command, the `require` built-in function, the `source` command, or the JavaScript `load` function. A path list determines the directories to search. The default is initialized from the value of the *APTLOADPATH* environment variable if set, otherwise it is the `packages` subdirectory of *`Arbortext-path`*, the `main` subdirectory of `packages`, and the `tools` subdirectory of `packages`.

You must include the system default packages in the load path to ensure proper operation of Arbortext Editor.

If there is an *`Arbortext-path`*`\custom\scripts` subdirectory at startup, the `\custom\scripts` is automatically prepended to the load path for ACL, JavaScript, JScript, and VBScript files. Putting these supported script file types in the `\custom\scripts` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

The path list is also automatically prepended to include any scripts added to the *`Arbortext-path`*`\custom\scripts` directory if you include `%D` in the path.

Example:

```
append_load_path("/packages")
```

Examples

```
set loadpath="c:\\Program Files\\Arbortext\\editor\\packages;
 c:\\Arbortext\\editor\\packages\\main"
```

Multiple directories are delimited with semicolons, and the entire set must be enclosed in quotation marks.

You can use the `append_load_path` function on page 222 to update the script load paths.

set localebackslash

`set` localebackslash= { on | off}

This set option controls the published rendering of the ASCII backslash character (\) when running in Japanese and Korean locales. (In the Japanese locale, the backslash appears as a yen. In the Korean locale, the backslash appears as a won.)

When using font styles and requesting a specific font, the backslash will be rendered in that font. Because the backslash is an ASCII character, it is usually classified as English. However, setting `localebackslash` to `on` causes the backslash to be rendered based on the current locale. Therefore, when running in a Japanese locale with `localebackslash` set to `on`, the backslash will be rendered in a Japanese font in published output, and will appear as a yen.

Setting `localebackslash` impacts the publishing of the backslash character as follows:

When running in the English locale:

- If `localebackslash` is set to `off`, classify as English.
- If `localebackslash` is set to `on`, classify as CJK. (Resolved to the value of localedefault on page 849 if is set. Otherwise, classify as Japanese.)

When running in any non-English locale:

- If `localebackslash` is set to `off`, classify as English.
- If `localebackslash` is set to `on` in a CJK locale, classify as the current system locale. (For example, Japanese, Korean, and so on.)
- If `localebackslash` is set to `on` in a non-CJK locale, classify as English.

set localedefault

`set` localedefault= { japanese | korean | simplifiedchinese | traditionalchinese}

This set option only applies if Arbortext Editor is running in English locale and if a font style, such as “serif” is specified, as opposed to a specific font. In this case, Arbortext Editor will try to classify a given Unicode character into a language.

`localedefault` has a default value of `japanese`. If a character is determined to be CJK but not assignable to a particular language, it will be assigned to this language. Also, certain Unicode blocks will be classified into this language if `localefavored` is set to `on`. The following Unicode blocks are affected:

02B0-02FF

0370-04FF

2000-20CF

2100-23FF

24FF-27BF

Example: The following setting causes the ideographs to be classified as Korean. Hangul will be classified as Korean in any case.

```
set localedefault=korean
```

set localefavored

`set` localefavored= { on | off}

This set option only applies if Arbortext Editor is running in English locale and if a font style, such as “serif” is specified, as opposed to a specific font. In this case, Arbortext Editor will try to classify a given Unicode character into a language.

When this set option is set to `on`, certain Unicode blocks will be considered to be characters in the locale indicated by `localedefault` rather than English. Default is `off`.

The following Unicode blocks are affected:

02B0-02FF

0370-04FF

2000-20CF

2100-23FF

24FF-27BF

set markupscan

`set` markupscan= { on | off}

This option lets you use search strings with the `find` command without specifying the `-markup` modifier. This option also specifies the default values for matching markup on the **Find** dialog box.

`off` is the default.

set menuaccelerators

`set` menuaccelerators= { on | off}

This command determines whether menu bar accelerators will be active. `on` is the default.

When set to `off`, you can still select a menu by pressing and releasing the **Alt** key and then typing the first letter of the menu name. (For example type `F` for the **File** menu.)

set messagelocation

`set` messagelocation= { statusbar | messagebox | either}

This command controls where error messages are displayed in the Arbortext Editor interface. When set to `statusbar`, error messages are always displayed in the status bar. In this case, messages will be truncated if necessary. When set to `messagebox`, error messages are always displayed in a separate message window. The default is `either`, which means to display error messages in the status bar if they fit, or in a message window otherwise.

```
set messagelocation=statusbar
set messageloc=messagebox
```

set modified

`set` modified= { on | off}

When this command is set to `on`, it marks the document as modified. When this command is set to `off`, it marks the document as not modified. In this case, the prompt to save unsaved changes when exiting or switching documents does not occur.

set modifyattrsdeleteempty

`set` modifyattrsdeleteempty= { on | off}

This option determines whether a CDATA attribute is deleted from a tag when you delete the last character of the attribute's value in the **Modify Attributes** dialog box. The default is `on` meaning to delete the attribute in this case. When set to `off`, deleting the last character of a CDATA attribute in the **Modify Attributes** dialog box results in the attribute value being set to an empty string. In this case, the value **(empty string)** appears for the attribute value in the **Modify Attributes** dialog box.

This option has global scope. It corresponds to the **Allow Empty String Attribute Values** option in the **Edit** category of the **Preferences** dialog box.

set modifyattrssorted

`set` modifyattrssorted= { on | off}

When this command is set to `on` (the default), Arbortext Editor alphabetically sorts the attribute names in the Modify Attributes dialog box.

When `modifyattrssorted` is set to `off`, Arbortext Editor sorts the attribute names for in the order they are defined in the (SGML) DTD. XML document attributes will remain alphabetized.

set movemode

`set` movemode= { on | off}

When this option is set to `on`, it enables a mode used to move or copy a highlighted selection using only the keyboard. When enabled, the insertion cursor is changed to the drag and drop insertion cursor and it can be moved away from the highlighted selection using normal cursor keys to the new cursor position. This mode remains in effect until either the **Enter** key is pressed (which accepts the move or copy) or **Escape** (which cancels it). The default is `off`.

By default, **F2** enables move mode and **Shift+F2** enables copy mode.

set msgfontpercent

`set` msgfontpercent= *n*

This command increases or decreases the point sizes of all fonts in all message windows to the percentage specified, subject to the availability of fonts. The default value is system dependent. Fonts cannot be displayed at less than 40% of their point size.

n is a number, excluding a percent (`%`) symbol.

set newlist

`set` newlist=*doctype_dir1* [;*doctype_dirn*]

This command specifies a list of paths to custom document type directories, which are used to populate the **Document Type** list in the **New Document** dialog box. It also enables you to remove a document type or a description of a document type from the list.

Note

You must enclose the value expression of this command in quotes. In general, any set command that takes an arbitrary string value must be quoted.

doctype_dir can be specified in the following formats:

- an absolute path to a DTD or a document type directory
- the base name of a document type directory

  Arbortext Editor searches for that directory based on the directory paths specified by the `set catalogpath` command.
- a path to a document type directory that is relative to one of the directory paths in the catalog path

Each document type is included in a category in the **New Document** dialog box. When a document type is included in the `set newlist` list, Arbortext Editor determines the document type's category. If the category has already been included, the document type is added to the end of that category list. If the category is included for the first time, then the category is added to the end of the category list and the document type is its first entry. The following rules are used to determine a document type's category:

- Use the name of the category configured in the document type configuration file (`.dcf`) file.
- Else, use the document type's application or custom name configured in the `application.xml` or `custom.xml` file in the `application` or `custom` directory.
- Else, use the name of the `custom` directory.

  If there are multiple `custom` directories that have the same name, then part of the path to the directories will be included to distinguish the directories.
- Else, use the **Other** category.

  Note that the **Other** category is always listed last in the **New Document** dialog box.

Use `%D` in the path to include the document types that are distributed with Arbortext Editor (*`Arbortext-path`*`\doctypes`). If you don't want to include the distributed document types, omit %D. You can specify document type directories before or after %D.

The order in which the document types appear in the list is determined by their order in the `set newlist` command. When you use %D, the distributed document types are listed in the following order in the **New Document** dialog box:

Category: **Technical Information Application**

Type:	Command Value:
• **Concept** • **Cover Page** • **Diagnostic** • **Procedure** • **Reference** • **Side by Side Procedure** • **Task** • **Topic** • **Troubleshooting** • **Map**	• *techinfo* (includes all Technical Information Application topic types) • *techinfo-rdedit* • *techinfomap*

Category: **DITA Technical Content**

Type:	Command Value:
• **DITA BookMap** • **DITA Map** • **DITA Concept** • **DITA Glossary** • **DITA Reference** • **DITA Task** • **DITA Topic** • **DITA Ditabase** • **DITA Key Definition Map** • **DITAVAL File**	• *dita\bookmap* • *dita\map* • *dita\ditabase* (includes all DITA topic types) • *dita\keybase* • *dita\ditaval*

Category: **DITA Learning and Training**

Type:	Command Value:
• **DITA Learning BookMap**	• *dita\learningBookmap*
• **DITA Learning Map**	• *dita\learningMap*
• **DITA Learning Assessment**	• *dita\learningDitabase* (includes all DITA Learning topic types)
• **DITA Learning Content**	
• **DITA Learning Overview**	
• **DITA Learning Plan**	
• **DITA Learning Summary**	

Category: **DocBook**

Type:	Command Value:
• **Arbortext XML DocBook V4.0**	• *axdocbook*
• **Arbortext Article (XML DocBook V4.0)**	• *asdocbook*

Category: **HTML**

Type:	Command Value:
• **HTML**	• *html*
• **XHTML**	• *xhtml*
• **HTML5**	• *html5*
• **XHTML5**	• *xhtml5*

Category: **Other**

Type:	Command Value:
• **Free-form XML**	• *freeform*
• **Text**	• *ASCII|Text*

Note

You cannot reorder the distributed document types in the **Document Types** list if you use the %D predefined value. Instead, you must specify each document type individually, in the order in which you want them to display in the **Document Types** list.

For example, if you want to add the `corpmemo` document type before the distributed document types and the `mydoctype` document type after the distributed document types in the **Document Type** list, set `newlist` to the following value:

```
set newlist="C:\\doctypes\\corpmemo;%D;mydoctype"
```

Note that the category in which these document types appear in the **New Document** dialog box is controlled by the category placement rules covered earlier.

You can also use the `set newlist` command to remove a distributed document type from the **Document Type** list. To do this, add a caret (^) before the document type that you want to remove.

For example, to remove the Arbortext XML DocBook V4.0 document type, set `newlist` to the following value:

```
set newlist="%D;^axdocbook"
```

Further, you can use the `set newlist` command to remove one or more descriptions of a document type from the **Document Type** list. Document type descriptions are defined in the DCF file associated with the document type. To remove descriptions, add a caret (^) before the document type with the description you want remove. Follow this with a pipe character (`|`) and the name of the description. You can add as many descriptions as you want, as long as a pipe character separates each description name. If the name of the description has a pipe character in its name, precede the pipe character in the name with another pipe character to escape the character in the description name.

For example, assume the `corpmemo` document type had two associated descriptions named `hrmemo` and `execmemo`. Set the `newlist` command to the following value: to remove those descriptions from the **Document Type** list:

```
set newlist="C:\\doctypes\\corpmemo;%D;^C:\\doctypes\\corpmemo|hrmemo|execmemo"
```

Note that in addition to using the `set newlist` command, you can also use the **New List Configuration** dialog box to control the contents of the **New Document** dialog box. You invoke the **New List Configuration** dialog box from the **Advanced preferences** dialog box.

The `newlist` option is automatically updated in your preferences file after you close a Arbortext Editor session in which you've changed the list of document types.

You can use the append_newlist_path on page 223 ACL function to update the document type paths that appear in the **Document Type** list in the **New Document** dialog box.

set objectboundarycolor

`set` objectboundarycolor= { *colorname* | *#rgbspec*}

This option determines the color used to display object boundary borders in the Edit window. The default value for this option is `gray5`. The value is either a named color or an RGB specification preceded by `#`. If `objectboundarycolor` is set to values of `inherit` or `default`, the boundaries are displayed with the default color of text in the Edit window (The default color is set with the `window_set` *canvasforeground* attribute or determined by the operating system default if otherwise not set.)

Examples:

```
set objectboundarycolor=inherit
set objectboundarycolor=blue
```

set openusesworkingdirectory

`set` openusesworkingdirectory= { on | off}

When the `openusesworkingdirectory` option is `on`, the **File ▸ Open** browser will display the current working directory whenever it is opened. By default, (corresponding to the `off` setting) the file browser remembers the last directory navigated (even across sessions) and returns to it the next time the browser is opened.

set othergraphicextensions

`set` othergraphicextensions= *extensions*

This option specifies a list of additional file extensions that are recognized as graphic files when inserting graphics using drag and drop. The extensions are also added to the default list of graphics file types displayed in the **Files of type** drop down list of the **Locate Graphic File to Reference** dialog box when inserting a new graphic.

extensions specifies a list of file extension patterns of the form *`*.ext1;*.ext2;*.ext3`*, separated by semi-colons.

Example

```
set othergraphicextensions="*.ppt;*.ico"
```

Arbortext Editor will not display such graphic files if they are not one of the supported graphic formats, unless an appropriate Active-X control is configured in the document type's `.dcf` file.

set outputlinebreak

`set` outputlinebreak= { unix | mswin | mac}

This option controls the line break characters written when a document or text file is saved by the `save` or `write` commands. When this option is set to , Arbortext Editor writes line breaks with CR/LF (Carriage Return, Line Feed) line delimiters, the normal ASCII coding for Windows. When this option is set to , Arbortext Editor writes text and SGML files with LF (Line Feed) line delimiters, the normal ASCII coding for UNIX platforms. When this option is set to , Arbortext Editor writes text and SGML files with CR (Carriage Return) line delimiters, the normal ASCII coding for Macintosh platforms. The default value is platform specific.

Arbortext Editor can read files saved in any of the formats.

set outputrecordlength

`set` outputrecordlength= { *n* | infinity}

This command specifies the preferred maximum length, *n*, of output lines written when saving a file. All characters (including tags and their delimiters) are counted against `outputrecordlength`. Lines that are longer than the `outputrecordlength` command setting are broken where a record end is not significant. Lines in "asis" sections (including SGML/XML comments) or lines ending with strings where line breaks may be significant (such as in attribute values or entity declarations) may exceed the `outputrecordlength` setting.

The minimum value for `outputrecordlength` is `40`. The default is `72`.

Setting `outputrecordlength` to `infinity`, causes no specific output record length to be set and no insignificant line breaks will be included in the selection or file, preventing line breaks in PCDATA.

You can specify the `-local` option (as part of the `set` command option list) to cause `outputrecordlength` to have a document-scope setting, only affecting the current document. Otherwise, `outputrecordlength` affects the session, meaning all documents use the current session value. You can override the current value by issuing a subsequent `outputrecordlength` setting.

`orl` is a synonym for `outputrecordlength`.

set overlaypagenumbers

`set` overlaypagenumbers= { on | off}

When set to `on`, Arbortext Editor will attempt to use pass reduction to speed the formatting documents. When set to `off`, it makes no attempt to use pass reduction. This affects all commands that format documents (`format`, `print`, `preview`). The default value is `on`.

Note

Setting the *APTNOOVERLAYPAGENUMBERS* environment variable disables pass reduction, thus overriding this setting.

set overlayunderflowtolerance

`set` overlayunderflowtolerance= { 0 | 1 | 2 | 3 | 4}

This option controls the tolerance for pass reduction. When pass reduction is enabled, this option sets the amount of disparity to accept when a final page variable value is smaller than the space it is overlaid in. Arbortext Editor never accepts an overlay if the new value is larger than the space it is overlaid on (the default is one character).

Note

Setting the *APTNOOVERLAYPAGENUMBERS* environment variable disables pass reduction, thus overriding this setting.

set pagebreaktext

`set` pagebreaktext=*string*

This command specifies the string to display centered in the horizontal rule used to denote a forced page break when breaks are displayed in the current tag display mode. If set to a null string, then only the horizontal rule is displayed. The default is `Forced Page Break`.

Examples

```
set pagebreaktext=""
set pagebreaktext="New Page"
```

set pagelayoutmarkers

`set` pagelayoutmarkers= { none | full}

This command controls how the page layout markup tags are displayed in the Edit view.

- `none` — Page layout markup is not displayed.
- `full` — All page layout markup is displayed.

This option is global. Page layout markup never appears in the document map.

The default setting is `none`.

The `caret` command is affected by the display of page layout markers. If page layout markers are displayed (`full`), they are counted during forward and backward motion.

For more detailed information on the page layout elements and their attributes, see www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

set papersize

`set` papersize= { default | usletter | uslegal | a4 | b5}

This command specifies the paper size setting for the print command. The `default` option uses the current printer's paper size setting to initialize the document in **Print Editor View**, **Print Published View**, and the **Print Preview ► Print** dialog boxes. If `set papersize` specifies one of the remaining four settings, then that specified setting is used.

set parserdeletespaces

`set` parserdeletespaces= { on | off}

When this option is `on`, the parser converts runs of white space in mixed content to a single space. This is useful for reading document instances written by other applications that add additional spaces, new-lines, and tabs intended to format the document when printed, even in adding mixed content where white space is significant. The default setting is `off`.

This option does not apply to free-form documents, asis sections, or xml:space= "preserve" sections.

set parservalidate

`set` parservalidate= { on | off}

This command alters the behavior of the parser used when an `edit -cc` is performed. This can be helpful when loading certain classes of files which are “out of balance”.

By default, all Arbortext Editor documents are normalized (that is, balanced). This means that every start tag has an end tag, no matter whether the DTD declares them as being optional or not. When you use Arbortext Editor to open a document that was authored outside of Arbortext Editor, Arbortext Editor will attempt to normalize the document by adding start and/or end tags. This is the behavior indicative of the `parservalidate` option being on. In some cases, these added tags may be unwanted, as the algorithm that adds them cannot always anticipate the author's original intent or compensate for a document that is well out of context. When the addition of start and end tags makes a document harder to work with (rather than easier), close the document (without saving changes), turn the `parservalidate` option off, and reload the document.

Be aware that the `parservalidate` option affects the document load process. It does not affect the editing or saving of a loaded document.

Note

If you load a document with the `parservalidate` option off, it is likely that the next **Check Completeness** operation will find several problems. This is because Arbortext Editor is expecting a balanced document, and with `parservalidate` off, the document was not balanced when it was opened.

The default is `on`.

set paste

`set` paste= { *buffername* | default}

This command creates a paste buffer of the name specified by *buffername* and makes it the current paste buffer. By default, the current paste buffer is a special buffer named `default`.

The current paste buffer is the buffer used by the `copy_mark`, `paste`, `delete_mark`, `read`, and `write` commands when no paste buffer name is specified.

Examples

```
set paste=seealso
set paste=Ifandonlyif
set paste=intropara
set paste=yrendtable
```

set pasteduplicateids

`set` pasteduplicateids= { prompt | ignore | remove | replace | show}

This command determines the action to be taken when duplicate IDs are created in a document during a paste action.

You can set the following values:

- `prompt` — invoke the Paste — Duplicate IDs dialog box.
- `ignore` — take no action, and write a warning to the status bar.
- `remove` — strip any duplicate IDs and ID references from the pasted content.

  Note: if the stripped attribute is required, a Completeness Check action for the document will report an error.
- `replace` — replace any duplicate IDs and ID references with generated unique IDs.
- `show` — invoke the IDs and ID References dialog box.

Examples:

```
set pasteduplicateids=replace
```

set pastegraphicspath

`set` pastegraphicspath = *dir*

This command determines the directory where graphics embedded in a Microsoft Word document are stored during a copy and paste operation. If you copy and paste part of a Microsoft Word document that contains embedded graphics, those graphics are not embedded in your XML document. Instead, they are removed from the Word document, stored on the file system, and referenced from your XML document. By default, the graphics are stored in a directory named `pastegraphics` in your Arbortext Editor Microsoft Windows application data directory. For example, if your Windows home directory is on the `C:` drive, the graphics would be stored in `C:\Document and Settings\`*`Your Login`*`\Application Data\PTC\Arbortext\Editor\pastegraphics`.

You can set *dir* to the following values:

- The full or relative path to a directory

  A relative path will be based on the current working directory. If the directory does not exist, it will be created.
- `%D` — Represents the default `pastegraphicspath` subdirectory of the application directory.
- `%B` — Represents the directory of the target XML document, if that directory exists and is writable.

  If the target directory does not exist or is not writable, the graphic is placed in the default location and the value of the graphic filename reference or entity reference uses the fully-qualified path.

Note that the pastegraphicspath directory is separate from the graphicspath directories. If you want the graphics in the pastegraphicspath directory to be generally available to Arbortext Editor, you must add them to the graphicspath directory list. If you add the pastegraphicspath directory to the graphicspath, then the converted graphic references will not include the directory path.

set pastenamespaceattrs

set pastenamespaceattrs= { on | off}

This option controls whether necessary namespace attributes are added when certain clipboard functions are performed. For example, consider the following document fragment:

```
<parent xmlns:ns="http://www.xyz.com">
<ns:child>text</ns:child>
</parent>
```

If a user copies the child element (<ns:child>text</ns:child>) and pastes that somewhere in the document (or any other Arbortext Editor document that has not declared the ns prefix), the resulting pasted fragment will have the namespace attributes automatically added, resulting in the following:

```
<ns:child xmlns:ns="http://www.xyz.com">text</ns:child>
```

The namespace attributes would not be added automatically if the same fragment was:

- Placed into a named paste buffer.
- Pasted into another application.

In you want the namespace attributes to be automatically added in these cases, you must set the option pastenamespaceattrs to on. The default is off.

set pastepreserve

set pastepreserve= { on | off}

This command controls the behavior of the paste command when the paste buffer contains tags or attributes which are not legal markup according to the target document's DTD. If this setting is *off*, such markup will be lost during a paste. If this setting is *on*, such markup will be preserved by inserting the non-legal tags and attributes as invalid markup.

The default value of pastepreserve is *off*.

set pastesource

set pastesource= [mif] [;htm] [;rtf] [;txt]

This command specifies the Microsoft Windows clipboard formats that will be converted to XML markup during Arbortext Editor copy and paste operations from other applications. The command also enables and disables the conversion of clipboard formats. To disable the conversion of clipboard formats during copy and paste operations from other applications, set this preference to an empty string.

The preference is set to some typical types of document formats that are stored on the Microsoft Windows clipboard when text is copied in an application other than Arbortext Editor, separated by semicolons (`;`). The `pastesource` preference has the following default value:

```
mif;htm;rtf;txt
```

The following values are supported:

- `mif` — Maker Interchange Format (MIF), the document format supported by Adobe FrameMaker
- `htm` — HTML markup
- `rtf` — Rich Text Format (RTF), the document format used by several Microsoft applications including Microsoft Word
- `text` — Unicode and 8-bit ANSI text

If a document format is removed from the default list, that format is no longer processed during conversion for copy and paste operations. However, note that many applications put more than one document format on the clipboard when text is copied. For example, Adobe FrameMaker puts both MIF and RTF formats on the clipboard when text is copied, so you could remove `mif` from the `pastesource` list and Arbortext Editor could still use the RTF version of the copied text for the conversion operation.

Examples:

```
set pastesource=""
set pastesource="rtf"
set pastesource="htm;rtf;txt"
```

set pdfconfigfile

`set` pdfconfigfile=*path-and-filename*

This option specifies the path and file name of the configuration file to use when publishing PDF files with the FOSI and XSL-FO engines.

If no path is specified, Arbortext Editor looks for the file in the path specified by the `set composerpath` on page 767 option.

You can 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.

You can also put a custom configuration file in *APTCUSTOM*\app*custom-file-name*.appcf (APP) or in *Arbortext-path*\custom\lib*custom-file-name*.pdfcf (FOSI), where it will be automatically accessible.

Example:
```
set pdfconfigfile=memo.pdfcf
```

Note

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

set pdfprinter

`set` pdfprinter=*printer*

This option specifies a Windows printer driver for producing PDF files during the current Arbortext Editor editing session.

You can also specify a `set pdfprinter` statement in a custom ACL file placed in *Arbortext-path*\custom\init*custom-file-name*.acl, where it will be loaded at start time.

Note

You only need to set this option if you are creating PDF output via Adobe Distiller. This is a deprecated method not recommended by PTC.

Example:
```
set pdfprinter=HP11PS
```

If the name includes spaces or special characters, it must be enclosed with quotes.

Note

Most PostScript printer drivers produce acceptable and reliable output for PostScript and PDF files.

You can get the Adobe Universal PostScript Windows Driver Installer from the Adobe web site at www.adobe.com.

The `set pdfprinter` option does not work for the Arbortext Publishing Engine. The default printer that the Arbortext Publishing Engine uses is configured using the **Arbortext Publishing Engine Configuration** program.

set pecompositionemail

`set` pecompositionemail= { *email_address* | ""}

This option specifies an email address that is transmitted to the Arbortext PE server with every publishing request. The empty string (the default) does not transmit an email address. The value for this option persists across sessions.

The Arbortext Publishing Engine global configuration parameter `com.arbortext.e3.compositionEmailPolicy` and a notifier application must be set up on the Arbortext PE server to use the email when sending notification. See your Arbortext Publishing Engine administrator or *Configuration Guide for Arbortext Publishing Engine* for information.

set pecompositionid

`set` pecompositionid= { *user_id* | ""}

This option specifies the user ID that is transmitted to the Arbortext PE server with a publishing request. The empty string (the default) does not transmit a user ID. The value for this option persists across sessions.

The value is used as the value of the HTTP query parameter name specified by the Arbortext PE server global configuration parameter `com.arbortext.e3.compositionIdentificationPolicy`. See your Arbortext Publishing Engine administrator or *Configuration Guide for Arbortext Publishing Engine* for information.

set pendingdelete

`set` pendingdelete= { on | off}

When `set pendingdelete` is set to `on` (the default), selected text is deleted when new characters or single tags are inserted (but only if the cursor is immediately to the left, right, or inside the selection). When this command is set to `off`, selected text must be explicitly cut or deleted.

set pequeuecomposition

`set` pequeuecomposition= { on | off}

This option specifies whether **Queue Transaction** is checked in the Arbortext Editor **File ► Publish** dialog boxes. When set to `on`, **Queue Transaction** is checked when the publish dialog boxes open. The default is `off`. The value for this option persists across sessions.

The `pequeuecomposition` option corresponds to the **Tools ► Preferences ► Publishing Engine** preference **Queue composition transactions**.

The ability to set **Queue Transaction** is available to Arbortext Editor users if the Arbortext PE server `queueCompositionOperations` global parameter is set to `optional`. The user can still clear **Queue Transaction** in the publish dialog boxes unless queuing is explicitly required to be on or off by the Arbortext Publishing Engine `queueCompositionOperations` parameter. In those cases, `pequeuecomposition` is ignored. See your Arbortext Publishing Engine administrator or *Configuration Guide for Arbortext Publishing Engine* for information.

set pequeuedeleteafterdownload

`set` pequeuedeleteafterdownload= { on | off}

This option specifies whether Arbortext Editor should automatically delete a transaction result from the **Tools ► Queued Transactions** viewer after the user downloads it. The default is `off`. The value for this option persists across sessions.

The `pequeuedeleteafterdownload` option corresponds to the **Tools ► Preferences ► Publishing Engine** preference **Delete transaction after downloading**.

set pequeuedeleteprompt

`set` pequeuedeleteprompt= { on | off}

This option specifies whether Arbortext Editor should prompt the user to confirm cancelling a queued transaction or deleting a transaction result from the **Tools ► Queued Transactions** viewer. The default is `on`. The value for this option persists across sessions.

The `pequeuedeleteprompt` option corresponds to the **Tools ► Preferences ► Publishing Engine** preference **Prompt before deleting a transaction**.

set pequeuedisplay

`set` pequeuedisplay= { on | off}

This option specifies whether the **Tools ► Queued Transactions** viewer is displayed after each Arbortext Editor queued publishing request. **Queue Transaction** must be checked in the Arbortext Editor **File ► Publish** dialog box to

treat the publishing request as a queued transaction. `pequeuedisplay` is ignored if a publishing operation is not queued. The default is `on`. The value for this option persists across sessions.

The `pequeuedisplay` option corresponds to the **Tools ► Preferences ► Publishing Engine** preference **Show transaction viewer after queuing**.

set pequeuedtransactionnames

`set` pequeuedtransactionnames= ['*type-specification*] [;*type-specification*] [...']

This option specifies the transaction name for a queued transaction being sent to Arbortext Publishing Engine.

type-specification specifies output type and transaction identifiers that can include a set of supported variables of the form:
`type:specification`

- *type*

  specifies the published output type. The type can be one or more of the following, separated by commas:

  - `epub` specifies EPUB
  - `html` specifies HTML File
  - `htmlhelp` specifies HTML Help
  - `pdf` specifies PDF
  - `preview` specifies Print Preview
  - `print` specifies Print
  - `rtf` specifies RTF
  - `usingxsl` specifies Using XSL
  - `web` specifies Web

  An asterisk (`*`) specifies all output types.

- *specification*

  is a string comprised of descriptive text and, optionally, any of the supported transaction name strings (`$d`, `$o`, `$t`, and `$u`).

Transaction name strings can be specified alone, with text, or in any combination, as part of the transaction name specification:

Transaction Name Strings

String	Description	Example
`$d`	Replaced by the name of the document being published.	`Preview:$d`
`$o`	Replaced by the output type.	`Print:$o-$d`
`$t`	The server replaces with the transaction ID.	`EPUB-doc-$d:$t`
`$u`	Replaced by the user ID, as specified by the `set pecompositionid` command on page 866 option.	`Dept123-$o-$u-$d`

Special escape characters may be needed to produce the desired result. For example, because a `;` is used as the *type*:*specification* separator, you would specify a semicolon in the transaction name using the special character combination `\;`.

- Specify a semicolon as part of a transaction name by entering `\;`.
- Specify a single quote as part of a transaction name by entering two single quotes.

Other special characters, such as `$`, `\`, `:`, or `,` may be coded as you would type them in a transaction name.

Examples:

```
set pequeuedtransactionnames='*:inventory_$t'
set pequeuedtransactionnames='pdf:$u-$d-$o;html,htmlhelp:$u-htmlpage-$d'
set pequeuedtransactionnames='pdf:''Docs\;$\:'''
set pequeuedtransactionnames='rtf:document\: my document'
```

set pequeueoverwritedirprompt

`set` pequeueoverwritedirprompt= { on | off}

This option specifies whether to prompt the user about overwriting an existing directory when retrieving and saving a published result from the **Queued Transactions** viewer (**Tools ▸ Queued Transactions**). Some publishing types return a file and a subdirectory of related graphics files, such as HTML, RTF, or XSL. When the user chooses an existing location for saving the file, the subdirectory name and contents are checked. The default is `on`, prompting the user to confirm overwriting files in an existing subdirectory that is not empty.

This option only applies if Arbortext Editor is using Arbortext Publishing Engine for publishing and has **Queue Transaction** checked for **File ► Publish ► HTML File**, **File ► Publish ► Using XSL**, or **File ► Publish ► RTF File**.

set peserverurl

`set` peserverurl= *URL*

To enable the Arbortext Editor client to communicate with the Arbortext Publishing Engine, specify a valid URL (HTTP or HTTPS) to the Arbortext PE Request Manager using `set peserverurl`. The `set peservices` on page 870 command must also be turned on.

Arbortext Editor retrieves information about the Arbortext Publishing Engine when Arbortext Editor starts. The returned information is stored in memory. There may be a waiting period associated with this process.

There is no default value. If you do not know your Arbortext PE Request Manager URL, ask your Arbortext Publishing Engine administrator.

In the following example, the URL specifies the server, port number, and the unique Arbortext Publishing Engine servlet specification as configured by the web server and servlet container where the Arbortext Publishing Engine is installed.
`http://PEserver.mycompany.com:8000/e3/servlet/e3`

Users can set this as a preference in Arbortext Editor. Choose the **Tools ► Preferences ► Publishing Engine** and enter the **URL**. They will also need to check **Use Publishing Engine**.

You can also place the `set peserverurl` command in a custom ACL file and put the file in the *`Arbortext-path`*`\custom\init` directory.

set peservices

`set` peservices= { on | off}

To enable the Arbortext Editor client to communicate with the Arbortext Publishing Engine, use `set peservices=on` and specify the URL to the Arbortext PE Request Manager using `set peserverurl` on page 870. Arbortext Editor retrieves information about the Arbortext Publishing Engine when Arbortext Editor starts. The returned information is stored in memory. There may be a waiting period associated with this process.

The default value is `off`. When `set peservices` is `off`, you can perform local publishing if your Arbortext Editor installation also has the appropriate licensing for Arbortext Styler, Print Composer, or Web/Wireless Composer.

Users can set this as a preference in Arbortext Editor. Choose the **Tools ► Preferences** and choose **Publishing Engine**. Check the **Use Publishing Engine** option from the list. They will also need to specify the URL following the form:

```
http://PEserver.mycompany.com:8000/e3/servlet/e3
```

You can also place the `set peservices` command in a custom ACL file and put the file in the *`Arbortext-path`*`/custom/init` directory.

set petransactionoptions

`set` petransactionoptions= { *name1:value1;name2:value2;...nameN:valueN* | ""}

This option specifies additional parameters that are transmitted to the Arbortext PE server with every publishing request. The empty string (the default) does not transmit any parameters. The value for this option persists across sessions.

name1:value1;name2:value2;...nameN:valueN is a series of *name*:*value* pairs separated by semicolons, where each *name* is separated from its *value* by a colon. To include a semicolon in a name or value, specify `\;`. To include a colon in a name, specify `\:`. To include a backslash in a name, specify `\\`.

See your Arbortext Publishing Engine administrator or *Configuration Guide for Arbortext Publishing Engine* for information.

set preferentityreference

`set` preferentityreference= { on | off}

For documents using a document type that supports inserting graphics as both file references and entity references, this option determines which attribute value is set when inserting graphics. In rare cases where a document contains a graphic element with both attributes set, this setting determines which attribute to use when displaying the graphic.

- `off` — (The default.) Use the file reference attribute.
- `on` — Use the entity reference attribute.

set prefersystemid

`set` prefersystemid= { on | off}

This command determines whether the system identifier (if `on`) or public identifier and entity name (if `off`) is given priority when searching a `catalog` file to resolve an entity. The default is `off`, meaning that the public identifier and entity name are preferred over the system identifier.

Note

If you set this preference to prefer system IDs, or set `prefer="system"` in your catalogs, be sure to have system entries map all http URIs to local files. For example, a document type declaration such as "<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">" would cause a request to be sent to the W3C servers if no system entry was provided for the URI.

set prefersystemidxmlcatalogs

`set` prefersystemidxmlcatalogs= { on | off}

This command determines whether the system identifier is given priority when searching an XML catalog file to resolve an entity (if set to `on`). The default is `off`, meaning that the public identifier and entity name are preferred over the system identifier.

Note

If you set this preference to prefer system IDs, or set `prefer="system"` in your catalogs, be sure to have system entries map all http URIs to local files. For example, a document type declaration such as "<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">" would cause a request to be sent to the W3C servers if no system entry was provided for the URI.

set preservereferencepaths

`set` preservereferencepaths= { on | off}

This option determines whether full file paths in the DITA reference attribute (usually the `href` attribute) are converted to relative paths during either an update references operation or the first save of a new DITA document. Set this option to `on` to preserve full reference paths and to `off` to convert full reference paths to relative paths. The default is `off`. This option has global scope.

Paths are only converted for file paths if a relative path can be determined from the current base URI, the DITA document directory, and any folders included in the DITA references path and graphics path. Paths are never converted for content management system (CMS) Logical IDs. Paths are also never converted for DITA references where the `scope` attribute is set to `external` or `peer`.

set printcolor

`set` printcolor= { on | off}

When you use Print Preview or Print Published with a black and white printer driver, you can `set printcolor=off` to print black and white instead of grayscale for color. The default is `on`.

set printeditorfooter

`set` printeditorfooter= { datemark | none | pageno | *"string"*}

This option specifies the content included at the bottom of a printed page when choosing **File ► Print Editor View**. The following options are available:

- *datemark* — Prints a datemark that identifies the document name, date, time, and user ID for the job.
- *none* — Suppresses the printing of footers. (The default footer.)
- *pageno* — Prints the ordinal page number.
- *”string”* — Allows you to specify a string delimited by a pair of single quotes (') or double quotes (”) to include in the footer.

set printeditorheader

`set` printeditorheader= { datemark | none | pageno | *"string"*}

This option specifies the content included at the top of the printed page when choosing **File ► Print Editor View**. The following options are available:

- *datemark* — Prints a datemark that identifies the document name, date, time, and user ID for the job. (The default header.)
- *none* — Suppresses the printing of headers.
- *pageno* — Prints the ordinal page number.
- *”string”* — Allows you to specify a string delimited by a pair of single quotes (') or double quotes (”) to include in the header.

set printeditorleftmargin

`set` printeditorleftmargin= *dimension*

This option specifies the indentation from the left edge of the paper to the point at which the text area is to begin. Enter the desired margin in the **dimension** argument. (For example, `8cm`, `5in`, `36pt`, and so on.) The default units is inches.

set printeditortopmargin

`set` printeditortopmargin= *dimension*

This option specifies the indentation from the top edge of the paper to the point at which the text area is to begin. Enter the desired margin in the **dimension** argument. (For example, `8cm`, `5in`, `36pt`, and so on.)

set printengineoverride

`set` printengineoverride= { no | app | fosi | xslfo}

This command allows you to specify the publishing engine for the current editing session, thus temporarily overriding settings made in a stylesheet's properties. The default value is `no`. Using a `no` value after a previous override action during the same session will return the environment to its effective print engine.

See *Logic for print/preview engine selection* in Arbortext Styler help for further information.

Note

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

set printer

`set` printer= { *printername* | default}

This command allows you to specify the default printer for the current editing session. The `default` setting restores the system default for printer name. If the name includes spaces or special characters, the name must be enclosed with quotes.

Examples

```
set printer=lw
set printer="HP LaserJet"
```

```
set printer=default
```

set printstylesheet

`set` printstylesheet= { *name* | none}

This command specifies the stylesheet to be used in place of the default stylesheet for published output (printing, print preview, and publish to PDF).

name is the path and file name of a FOSI, XSL, or Arbortext Styler stylesheet. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, then a `.fos` file, and finally a `.xsl` file.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:
  ```
  set printstylesheet=D:\ArbortextUser\axdocbook.fos
  ```
- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.
  ```
  set printstylesheet=axdocbook.fos
  ```

If `none` is specified, or if the `set printstylesheet` command is not set, then the stylesheet specified by the `set stylesheet` command is used for published printing and print preview, as well as for the Editor view. Arbortext Publishing Engine looks for a stylesheet with a matching name.

Examples

```
set printstylesheet=memo2.style
set printstylesheet=~/mydocs/memo/memo2.fos
set printstylesheet=http://www.some_site.com/examples/techman.xsl
```

set promptattrs

`set` promptattrs= { on | off}

When set to `on`, this command opens the **Modify Attributes** dialog box whenever an element having attributes is inserted. The default is `off`.

set promptentitydir

`set` promptentitydir= { doc | current | entitypath}

This command sets the default location for the starting directory in two dialog boxes:

- The **Locate File to Reference** dialog box displays a starting directory where you can choose a file entity. Selecting **Insert ► File Reference** from the Arbortext Editor menu brings up the **Locate File to Reference** dialog box.
- The **Assign System ID** dialog box displays a starting directory where you can choose a file entity. Selecting **Entities ► File** from the Arbortext Editor menu brings up the **File Entities** dialog box. Selecting the **Browse** button next to the **System ID** field brings up the **Assign System ID** dialog box.

The `doc` setting is the path for the current document. The `current` setting is the current working directory for Arbortext Editor. The `entitypath` setting is the first path specified by the **Entities Path** preference. To find the **Entities Path** preference, select **Tools ► Preferences**, and then select the **File Locations** category.

The default setting is `doc`.

set promptgraphicbrowser

`set` promptgraphicbrowser= { on | off}

This command opens a file selector dialog box when a graphic tag is added. The default is `on`.

When `set promptgraphicbrowser` is `off`, use attribute settings to control file selection.

set promptgraphicdir

`set` promptgraphicdir= { doc | current | graphicspath}

This command sets the default location for the starting directory in two dialog boxes:

- The **Locate Graphic File to Reference** dialog box displays a starting directory where you can choose a graphic file. Selecting **Insert ► Graphic** from the

Arbortext Editor menu brings up the **Locate Graphic File to Reference** dialog box.

- The **Assign System ID** dialog box displays a starting directory where you can choose a graphic entity. Selecting **Entities ► Graphic** from the Arbortext Editor menu brings up the **Graphic Entities** dialog box. Selecting the **Browse** button next to the **System ID** field brings up the **Assign System ID** dialog box.

The `doc` setting is the path for the current document. The `current` setting is the current working directory for Arbortext Editor. The `graphicspath` setting is the first path specified by the **Graphics Path** preference. To find the **Graphics Path** preference, select **Tools ► Preferences**, and then select the **File Locations** category.

The default setting is `doc`.

set promptgraphictags

`set` promptgraphictags= { on | off}

This command specifies whether Arbortext Editor prompts users for a choice of graphics tags when they insert a graphic at a location where more than one graphic element is valid. The default is `on`. Graphic tags are identified in the `.dcf` file.

When this option is set to `off`, Arbortext Editor automatically uses the primary graphic tag.

set promptnodtd

`set` promptnotdtd= { on | off}

When a document is loaded for which the DTD or Schema cannot be found and `promptnodtd` is set to `on`, the user is prompted whether to perform one of the following actions:

- Browse for the DTD/Schema.
- Edit the document in free-form XML mode.
- Edit the document as an ASCII document.

If `promptnodtd` is set to `off`, the use can only edit the document as an ASCII document. A document instance without a DOCTYPE declaration will be opened in free-form XML mode, without prompting if `promptnodtd` is set to `off`.

The default value of `promptnodtd` is *on*.

set promptstylesheetassociations

`set` promptstylesheetassociations= { on | off}

This command options determines whether to display a dialog box warning the user that a stylesheet association in the document being published will be ignored. The default is `on`, which displays the warning. At the warning prompt, you can choose to turn the option `off`. The setting is saved and no further warnings are displayed.

This option applies when publishing documents using the Arbortext Publishing Engine, the stylesheet association applies to the type of publishing requested, and the stylesheet association specifies a stylesheet by file path rather than by URL.

If a document contains a stylesheet association specified with a path, Arbortext Publishing Engine will attempt to locate a stylesheet of the same name on the Arbortext PE server. The path in the stylesheet association will be ignored, and only the stylesheet name is used by the server to determine a match.

set prompttablemodels

`set` prompttablemodels= { on | off}

When set to `on` (the default), this command prompts users for a choice of table models when they insert a table if the cursor is in a location in which it is valid to insert more than one table model.

When this option is set to `off`, Arbortext Editor uses the default table model, which is set in the `.dcf` file.

set protection

`set` protection= { write | none}

This setting allows you to change the protection on specific regions within a document type.

You can protect a document type in the `.dcf` file, which means you cannot edit documents that use the document type. However, you can also specify tags that are not protected in the `.dcf` file, allowing you to edit content within these tags. The protected status you set remains constant for tags in all instances of this document type. You can also restrict editing for individual elements in a document type in the `.dcf` file.

When `set protection` is `write` (the default), protected regions are displayed, but editing is prevented.

When set to `none`, the protection is overridden, and protected regions are displayed and can be edited.

set protectpagelayout

`set` protectpagelayout= { on | off}

This command controls whether page layout markup tags are read only or if they can be added, deleted, or modified.

- `on` — Page layout markup is read only and cannot be deleted.
- `off` — Page layout markup may be modified.

This option is global. The default setting is `on`. It does not affect the `delete_mark` (cut) command.

For more detailed information on the page layout elements and their attributes, see www.arbortext.com/namespace/pagelayout. Refer to the *Programmer's Reference* for more detailed information about line numbering applications.

set quicktags

`set` quicktags = { on | off}

When `set quicktags` is set to `on`, a list of valid tags appears when you press **Enter** while editing. When `set quicktags` is set to `off`, the list of valid tags does not appear when **Enter** is pressed.

If another action (such as **Smart Insert**) is mapped to the **Enter** key, you can use the **Enter** key on the numeric keypad.

set reportinvalidmarkup

`set` reportinvalidmarkup= { on | off}

This option controls whether invalid markup is reported as an error during the loading of a file. Even if this is set to `off`, invalid markup will still be reported when choosing **Tools ► Check Completeness**.

The default is `on`.

set requireattrs

`set` requireattrs= { on | off}

When `set requireattrs` is set to `on` and a tag that has required attributes is inserted, the **Modify Attributes** dialog box for that tag is automatically opened. If you attempt to exit or quit the **Modify Attributes** dialog box without supplying values for all required attributes, you are asked if you wish to supply required attributes before exiting.

If inserting the tag causes other tags with required attributes to be inserted, you are presented with the **Modify Attributes** dialog box for those tags as well.

set revertfocus

`set` revertfocus= { on | off}

This option is used by dialog windows to switch input focus from the dialog to another window, most likely the user instance window that is the parent of the dialog box.

Currently, its only use will be by the insert markup dialog box. If this option is set to “on” and the user selects markup from the dialog, the focus will be switched back to the user instance with the caret at the insertion point. Otherwise, the input focus will remain on the dialog box.

set rochange

`set` rochange= { on | off}

If set to `on`, this command allows you to make changes to a read-only document, although you will only be able to save these changes with the `write` or `save_as` command. `off` is the default.

This command is disabled in the *original* view of a document that contains change tracking markup.

set rowrulerunit

`set` rowrulerunit= { in | cm | mm | pt | pc | pi}

This command sets the units for the Row Ruler. Available units are:

- `in` — inches (default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pc` — picas
- `pi` — picas

set rtfpreview

`set` rtfpreview= { on | off}

This option controls whether the resulting RTF document is displayed following a Publish to RTF action instigated by the **File ► Publish ► RTF File** menu option. When this option is set to `on`, Arbortext Editor will display the resulting RTF file in Word (or WordPad if MS Word is not installed). If you set the preference to `off`, the **Publish to RTF** dialog box will show the **View RTF** option unchecked. If

you subsequently check the option in the dialog, the RTF document will be displayed automatically following the publish, but because you have specifically set the option to `off` in your preferences, the value will be unchecked each time the dialog box is opened. When the `rtfpreview` option is set to `off`, Arbortext Editor will make no attempt to display the RTF file, which is important if there is no applicable RTF reader available.

set rtfstylesheet

`set` rtfstylesheet= { *name* | none}

This command specifies the stylesheet to be used as the default for the **Publish ▸ RTF File** option on the **File** menu.

name is the path and file name of a `.style` or `.xsl` stylesheet file. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, and then a `.xsl` file.

The stylesheet ID processing instruction must specify `rtf` as the publishing type.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:
  ```
  set rtfstylesheet=D:\ArbortextUser\axdocbook.style
  ```
- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.
  ```
  set rtfstylesheet=axdocbook.style
  ```

If `rtfstylesheet` is set to `none` or is not set, Arbortext Editor and Arbortext Publishing Engine use the Editor/Default stylesheet.

Examples

```
set rtfstylesheet=memo2
set rtfstylesheet=~/mydocs/memo/memo2.style
set rtfstylesheet=http://www.some_site.com/examples/techman_web.xsl
```

set saverenames

`set` saverenames= { on | off}

When this option is set to `on` (the default), the `save` on page 704 command saves a document by writing a temporary file, deleting the original file, and then renaming the temporary file to the original name. When this option is set to `off`, `save` rewrites the original file directly. The default is `on` to prevent data loss in the case of insufficient disk space. However, the rename strategy may sometimes fail on certain kinds of network file systems. For those situations, set `saverenames` to `off`.

set savewindowconfiguration

`set` savewindowconfiguration= { yes | no | prompt}

This option determines whether Arbortext Editor saves certain current window and view settings when it exits. This option has a global scope. The settings are:

- `yes` — Save the settings.
- `no` — Do not save the settings.
- `prompt` — Prompt the user to decide whether settings should be saved. This is the default setting.

The size and location of all modeless dialog boxes that have been opened during the current session will be saved.

The `set` command options saved by this command are the menu, toolbar, split window, and window position settings.

Window settings

Description	set command option
Window geometry	none
Split screen	`set viewmode` on page 924
Percent of split screen for the Document Map or Column view pane	`set docmappercent` on page 785
Where the Document Map or Column view pane is located in a split screen	`set docmapside` on page 786
Whether the Document Map or Column view pane and the Edit pane	`set docmapsync` on page 786

Window settings (continued)

Description	set command option
are synchronized	
Toolbar location, whether docked or floating	none
Docked dialog boxes and their locations	none

Note

The split screen setting only applies to one Edit pane and one Document Map or Column view pane. If the saved state has a split window with the same views , then the setting will restore only one view.

View settings used by Edit and Document Map panes

Description	set command option
Show generated text	`set gentext` on page 815
Auto-update mode	`set gentextautoupdate` on page 816
Expand entities (document scope)	`set showentities` on page 887
Change tracking view	`set viewchangetracking` on page 923

If one of the preceding option values differs between the Edit pane and the Document Map pane, then the value for the current view is used.

Edit pane settings

Description	set command option
Text font scale percent	`set editfontpercent` on page 789
Show marked section status	`set showmsstatus` on page 889
Show ignore sections	`set showignorems` on page 888
Show graphics	`set graphicdisplay` on page 823
Show links	`set showlinks` on page 888
Show tables	`set tabletags` on page 905
Tag display	`set tagdisplay` on page 907
Tag display in table cells	`set tabletagdisplay` on page 905

Edit pane settings (continued)

Description	set command option
Table rulers	`set tablerulers` on page 904
Ruler units	`set columnrulerunit` on page 766, `set rowrulerunit` on page 880

Document Map pane settings

Description	set command option
End tag display	`set docmapendtags` on page 783
Text display	`set docmaptextdisplay` on page 786
All attributes expanded (document scope)	`set docmapshowattrs` on page 785

set selectionsvc

`set` selectionsvc= { on | off}

If set to `on` (the default), this command allows you to paste between the Edit window and other windows on the workstation. If set to `off`, Arbortext Editor has a separate paste buffer, the contents of which are unaffected by cutting or copying from other windows.

set selectscan

`set` selectscan= { on | off}

When set to `on` (the default), the `find` command highlights the string it locates. Otherwise, the cursor is positioned after the string.

set sgmlextension

`set` sgmlextension= *ext*

This command specifies the extension to use for newly saved SGML files. `sgm` is the default. The **Save As** dialog box and `save_as` command add this extension if the specified file does not already have an extension and the option is non-null.

Note, a period should not be included as part of the extension.

Examples

```
set sgmlextension=sgm
set sgmlextension="sgml"
```

set sgmlselection

`set` sgmlselection= { on | off}

The `off` option for this command allows you to paste between windows in regular untagged ASCII format. If set to `on` (the default), the contents are pasted in tagged format. This is necessary if you are pasting tags, entities, or tables from one window to another.

set showattrs

`set` showattrs= { on | off}

When this command is set to `on` (the default) attributes are listed after the tag name as part of the Edit window tag display.

set showbreaksfulltags

`set` showbreaksfulltags= { on | off}

This command enables or disables the display of FOSI specified forced page and column breaks when the tag display is full. The default is `off`.

set showbreaksnotags

`set` showbreaksnotags= { on | off}

This command enables or disables the display of FOSI specified forced page and column breaks when the tag display is none. The default is `on`.

set showbreakspartialtags

`set` showbreakspartialtags= { on | off}

This command enables or disables the display of FOSI specified forced page and column breaks when the tag display is partial. The default is `off`.

set showcomments

`set` showcomments= { on | off}

This command enables or disables the display of comments in the Edit window and the Document Map. If your cursor is in the Edit window and `set showcomments` is set to `off`, comment text in the Edit window is hidden and barbell icons represent the comment tags if the tag display mode is **full** or **partial**. If the tag display mode is **none**, no markers appear in the Edit window.

If your cursor is in the Document Map and `set showcomments` is set to `off`, the comment icon, label, and text are hidden in the Document Map.

The default setting is `on`.

set showconrefs

`set` showconrefs= { on | off}

This command enables or disables the display of content references in the Edit window and the Document Map. If `set showconrefs` is set to `off`, referenced content is not displayed in the Edit window. If `set showconrefs` is set to `on`, the content of the element is replaced by the resolved value of the content reference attribute, displayed as generated text.

Display of generated text must be enabled to show content reference content.

The default setting is `on`.

set showcursor

`set` showcursor= { on | off}

When this command is set to `on`, the cursor is visible in the view. The default value is `on` except for the `dir` view.

set showdashedlines

`set` showdashedlines= { on | off}

If this option is set to `on` (the default), then the lines between elements in the Document Map are dashed. If set to `off`, then the lines are solid.

set showdetail

`set` showdetail= { on | off}

If this option is set to `on` (the default), then detailing is displayed in the current window as set by the `detail` command. If set to `off`, then the document is full expanded, regardless of the detailing set for each element.

Note, this command has no effect in the Document Map view.

set showemptyelement

`set` showemptyelement= { on | off}

This command allows you to control the gray block with the element name in it when there's an element without any content. If this option is set to `on` (the default), then an element with no content will be displayed as a gray box containing the start tag name. If the option is set to `off`, then the empty element box is not displayed.

Note

This command only applies when the `set tagdisplay` on page 907 is set to `none` and has no effect in the Document Map view.

set showentities

`set` showentities= { none | file | text | all}

This command determines whether file and text entities are displayed in the Edit window document. When set to `none`, no file or text entities are displayed. When set to `file` or `text`, the file or text entities are displayed respectively. If set to `all`, this command displays both types of entities. The default is `all`.

set showiconsfulltags

`set` showiconsfulltags= { on | off}

This command enables or disables the display of icons in the Edit pane when the tag display is set to full tags. The icon display is view local scope. The default setting is `on`.

The command does not affect the display of icons in the Document Map. Also, the display of icons within a table cell depends on the tag display set within the table through the `set tabletagdisplay` on page 905 command.

The following icons are affected by this command:

- File entity icons
- Product icons
- Icons set using the `oid_set_icon` on page 456 function

set showiconsnotags

`set` showiconsnotags= { on | off}

This command enables or disables the display of icons in the Edit pane when the tag display is set to no tags. The icon display is view local scope. The default setting is `off`.

The command does not affect the display of icons in the Document Map. Also, the display of icons within a table cell depends on the tag display set within the table through the `set tabletagdisplay` on page 905 command.

The following icons are affected by this command:

- File entity icons
- Product icons
- Icons set using the `oid_set_icon` on page 456 function

set showiconspartialtags

`set` showiconspartialtags= { on | off}

This command enables or disables the display of icons in the Edit pane when the tag display is set to partial tags. The icon display is view local scope. The default setting is `on`.

The command does not affect the display of icons in the Document Map. Also, the display of icons within a table cell depends on the tag display set within the table through the `set tabletagdisplay` on page 905 command.

The following icons are affected by this command:

- File entity icons
- Product icons
- Icons set using the `oid_set_icon` on page 456 function

set showignorems

`set` showignorems= { on | off}

This command enables or disables the display of marked sections with an IGNORE status. If `off`, IGNORE status sections are hidden. Barbell icons represent hidden marked sections if the tag display is full or partial. If the tag display mode is none, no markers appear. The default is `on`.

set showlinks

`set` showlinks= { on | off}

This command enables or disables the display of the `_link` tag pair and `_target` tag icon. The default is `on`.

set showmsgnum

`set` showmsgnum= { on | off}

If this option is set to `on` (the default), then error and informational messages displayed in the status line and various dialog boxes are prefixed with a unique message number on the form `[Annnnn]`. If set to `off`, then message numbers are not displayed.

This message number may be needed by Arbortext Technical Product Support when reporting error situations, particularly when error messages have been customized at your site to display non-standard messages.

set showmsstatus

`set` showmsstatus= { on | off}

If this command is set to `on` (the default), marked section status keywords are displayed in the `_include`, `_ignore`, `_cdata`, or `_rcdata` tags used to denote marked sections. If set to `off`, then only the tag name appears.

set shownamespaceattrs

`set` shownamespaceattrs= { on | off}

This command controls the display of namespaced attributes. When set to `on` (the default), this command displays the attributes of namespaced elements in the Editor view. When set to `off`, the attributes of namespaced elements are suppressed in the Editor view.

set shownamespaceprefix

`set` shownamespaceprefix= { on | off}

This command enables or disables the display of namespace prefixes in the Edit pane and Document Map. If this option is set to `off`, all elements are displayed without namespace prefixes.

The option has view scope and the default setting is `on`.

set shownewlines

`set` shownewlines= { on | off}

If this command is set to `on` (the default), a newline character will display in asis sections. If set to `off`, the newline character will not display.

set showobjectboundaries

`set` showobjectboundaries= { on | off}

This option determines whether object boundary borders are displayed in the Edit window. The default value for this option is `on`. This option has view scope.

set showpastewindow

`set` showpastewindow= { on | off}

The command determines whether the **Invalid Paste Structure** dialog box opens when a copy and paste operation from another application to Arbortext Editor fails due to context errors in the converted XML markup. The default is `on`, meaning that if the markup you are trying to paste is not valid at the paste location, the **Invalid Paste Structure** dialog box opens. This dialog box enables you to view the markup that Arbortext Editor is trying to paste into the document. The dialog also enables you to copy and paste all or part of that markup into your document.

The **Invalid Paste Structure** dialog box has a **Do not show this window when paste results are invalid** check box. When this check box is selected, the `showpastewindow` preference is set to `off` and the dialog box no longer appears when pasting converted XML markup would result in context errors. In this case, a message appears on the status bar saying that the paste operation would result in a context error.

Examples:

```
set showpastewindow=on
```

set showprelimuserules

`set` showprelimuserules= { on | off}

This command controls if the preliminary index (non-zero userules) is displayed in the editor view. The `on` setting displays the preliminary index.

The default is `off`.

set showprofileshading

`set` showprofileshading= { on | off}

This command enables or disables profile and inline applicability shading in the Edit windows. The default is `off`.

Profile shading is configured in the profile configuration file (`.pcf`) for the document type.

Inline applicability shading is presented in a single color. The color is defined with the `set inlineapplicabilitycolor` option for your application.

set showscreenhiddenattrs

`set` showscreenhiddenattrs= { on | off}

This command controls the display of attributes configured in the document type configuration file (`.dcf`) to be hidden in the Edit and Document Map panes. When set to `off`, Arbortext Editor doesn't display hidden attributes. When set to `on`, the `.dcf` file setting is ignored, and Arbortext Editor displays hidden attributes. The default setting is `off`.

set showspaces

`set` showspaces= { on | off}

This command controls the appearance of space and tab characters. When `showspaces` is `on`, Arbortext Editor displays a character for each space or tab in text content:

- Spaces: a small dot above the baseline
- No break spaces: the degree/ring character (°)
- Tabs: a right arrow (⇒)

The default is `off`.

set showunknownattrs

`set` showunknownattrs= { on | off}

This command controls whether attributes which were not declared are displayed or suppressed. Such attributes are a form of invalid markup.

The default is `on`.

set showxmlnsattrs

`set` showxmlnsattrs= { on | off}

This command controls whether namespace declaration attributes (XML attributes of the form `xmlns` and `xmlns:`*`prefix`*, where *prefix* is any valid XML markup name) are displayed in the Edit and Document Map views. The default is `on` and the option has view scope.

Note

The option `set shownamespaceattrs` controls whether foreign namespaced attributes are displayed. That includes all attributes on a foreign namespace tag or attributes of the form *`prefix:name`* where *prefix* corresponds to a declared, foreign namespace. The existing option `set showunknownattrs` controls whether undeclared attributes are shown.

- `set shownamespaceattrs` on page 889
- `set showunknownattrs` on page 891

set skipautosavecheck

`set` skipautosavecheck= { on | off}

When `set skipautosavecheck` is set to `on`, Arbortext Editor does not check for the presence of autosave or crash files when opening a file. Autosave and crash files are still created, if that functionality is enabled, but are not used to open a file.

The default setting is `off`.

set skipinlineelements

`set` skipinlineelements= { on | off}

The `set` option and advanced preference `skipinlineelements` controls whether inline elements are treated as word boundaries when checking spelling. When set to `on` (the default), the spelling checker will ignore inline element boundaries when evaluating the spelling of words. When set to `off`, the spelling checker treats each inline element as a word boundary.

The preference has document scope.

The setting of `skipinlineelements` may be overridden depending on the setting of the `.dcf` file entry `spellCheckingNewWord` for the document type. `spellCheckingNewWord` has the following possible values:

- `yes` — An element will always start a new word, regardless of the setting of the `skipinlineelements` preference.
- `no` — An element will not be considered a word boundary, regardless of the setting of the `skipinlineelements` preference.
- `default` — (The default setting.) An element will follow the setting of the `skipinlineelements` preference.

set smartinsert

`set` smartinsert= { on | off}

When `set smartinsert` is set to `on`, **Smart Insert** is enabled if it is configured in the document type's `.dcf` file. When `set smartinsert` is set to `off`, Arbortext Editor uses the current cursor location when inserting elements.

The default setting is `off`. Note that the `set quicktags` on page 879 setting can be affected by the setting of `set smartinsert`.

set spellabsoluteaddresses

`set` spellabsoluteaddresses= { on | off}

When this command is set to `off`, the spell checker ignores absolute URL's (tokens starting with `http://`, `https://`, `file://`, `ftp://`, and `mailto:`), absolute file paths (tokens that begin with `/`, `\\`, or *`X`*`:\`), and e-mail addresses (tokens that contain a `@` and a `.`). When set to `on`, the spell checker will flag those tokens as misspellings. The default is `off`.

set spellalphanums

`set` spellalphanums= { on | off}

When this command is set to `on`, the spelling checker flags words containing combinations of alphabetic and numeric characters; for example, ten4 or dos2unix. If this option is `off`, words containing numbers are ignored. The default is `off`.

set spellinteractive

`set` spellinteractive= { on | off}

When this command is set to `on`, Arbortext Editor automatically checks your document for spelling errors and marks any suspect words with a red wavy underline. The default is `on`.

set spellnumerals

`set` spellnumerals= { on | off}

When this command is set to `on`, the spell checker flags all Arabic and Roman numerals in the document. The default is `off`.

set spellrepeatword

`set` spellrepeatword= { on | off}

When this command is set to `on`, the spell checker flags words occurring twice in succession, such as the the. The default is `on`.

set spellsentencecapitalization

`set` spellsentencecapitalization= { on | off}

If this command is set to `on`, the spell checker flags words following a terminal punctuation mark (such as a period) that are not capitalized. The default is `off`.

set spellskiptags

`set` spellskiptags= { on | off}

When this command is set to `on`, the content in elements defined in the document type's `.dcf` file is skipped during a spelling check. The default is `on`.

set stricterrors

`set` stricterrors= { on | off}

When `set stricterrors` is set to `on`, a reference to an undeclared ACL variable in a package declared with the *_STRICT_* global variable will be flagged as an error. Arbortext Editor will terminate the ACL script at the point at which the undeclared variable is referenced. If set to `off`, a warning message is sent to the **Debug Messages** window if `set debug = 1`. Otherwise, no warning or error is generated.

set stylerconfirmdeletes

`set` stylerconfirmdeletes= { on | off}

When set to `on` (the default), users are prompted whenever they delete an object such as an element, context, condition, or page set in Arbortext Styler.

This command corresponds to **Options ► Confirm Deletions** in Arbortext Styler.

set stylercontextformatxsl

`set` stylercontextformatxsl= { on | off}

When set to `on`, Arbortext Styler displays contexts in XSL syntax. When set to `off` (the default), Arbortext Styler displays contexts in a text format.

For example, if you create a context for `title` within `chapter`, the XSL version displays as **chapter/title**, and the text version displays as **title in chapter**.

This command corresponds to **Options ► Show Contexts as XSL** in Arbortext Styler.

set stylerdocelementsonly

`set` stylerdocelementsonly= { on | off}

When set to `on` (the default), Arbortext Styler displays in the **Elements** list all of the elements currently used in the document. When set to `off`, Arbortext Styler displays all elements in the stylesheet in the **Elements** list, which typically corresponds to all elements in the DTD or schema.

Setting this command to `on` is the equivalent of activating the **View ► Only Elements in Document** menu option in Arbortext Styler.

set stylerdoctypeelementsonly

`set` stylerdoctypeelementsonly= { on | off}

When set to `off` (the default), Arbortext Styler displays all elements in the stylesheet in the **Elements** list. When set to `on`, Arbortext Styler displays in the **Elements** list the elements from the stylesheet that belong to the current document type.

Setting this command to `on` is the equivalent of activating the **View ► Only Elements in Document Type** menu option in Arbortext Styler.

set stylererrorcolor

`set` stylerrrorcolor= { *colorname* | *#rgbspec*}

This command determines the font color used in error messages displayed throughout Arbortext Styler. The value is either a named color or an RGB specification preceded by `#`. The default is `red`.

Examples (the following are equivalent):

```
set stylererrorcolor=red
set stylererrorcolor=#ff0000
```

set stylerexplicitfontcolor

`set` stylerexplicitfontcolor= { *colorname* | *#rgbspec*}

This command determines the font color used to display the labels of properties that have been explicitly set in the Arbortext Styler window or dialog boxes. This setting also applies to the color of the property category‘s label when a property from that category is explicitly set. The value is either a named color or an RGB specification preceded by `#`. The default is `blue`.

Examples (the following are equivalent):
```
set stylerexplicitfontcolor=red
set stylerexplicitfontcolor=#ff0000
```

set stylergentexttagfontcolor

`set` stylergentexttagfontcolor= { *colorname* | *#rgbspec*}

This command determines the font color of the generated text (gentext) tag in the Arbortext Styler generated text edit window. The value is either a named color or an RGB specification preceded by `#`. The default is `brown`.

The generated text tag also has a background color that is configured using the `set stylergentexttagshading` on page 896 setting.

Examples (the following are equivalent):
```
set stylergentexttagfontcolor=red
set stylergentexttagfontcolor=#ff0000
```

set stylergentexttagshading

`set` stylergentexttagshading= { *colorname* | *#rgbspec*}

This command determines the font shading of the generated text (gentext) tag in the Arbortext Styler generated text edit window. The value is either a named color or an RGB specification preceded by `#`. The default is `yellow`.

The generated text tag also has a font color that is configured using the `set stylergentexttagfontcolor` on page 896 setting.

Examples (the following are equivalent):
```
set stylergentexttagshading=red
set stylergentexttagshading=#ff0000
```

set stylerhassourceeditsfontcolor

`set` stylerhassourceeditsfontcolor= { *colorname* | *#rgbspec*}

This command determines the font color of the text confirming the presence of edited source for an element, in the Description tab of the Arbortext Styler window. The value is either a named color or an RGB specification preceded by `#`. The default is `orange`.

Note that the setting will not affect the edited source symbols in the list view window, only the heading of the Description tab and the text advising you that source edits exist for the element for the particular output source.

You can set this option as a preference from the **Tools ► Preferences** dialog box. Click the **Advanced** button and choose `stylerhassourceeditsfontcolor`.

Examples (the following are equivalent):

```
set stylerhassourceditsfontcolor=red
set stylerhassourceeditsfontcolor=#ff0000
```

set stylerhtmlversionoverride

`set` stylerhtmlversionoverride= { html | xhtml | html5 | xhtml5 | no}

This command allows you to specify the version of HTML to be output during a publishing operation, temporarily overriding settings made in a stylesheet's properties. The default value is `no`. Using a `no` value after a previous override action during the same session will revert the HTML version to that specified by the stylesheet.

set stylerindeterminatefontcolor

`set` stylerindeterminatefontcolor= { *colorname* | *#rgbspec*}

This command determines the font color used to display the label in the Arbortext Styler window when the value of the field is inherited from multiple sources and the values vary. This may occur, for example, for a **Size** field when multiple contexts are selected for a title element and the font size varies between contexts. The value is either a named color or an RGB specification preceded by `#`. The default is `orange`.

Examples (the following are equivalent):

```
set stylerindeterminatefontcolor=red
set stylerindeterminatefontcolor=#ff0000
```

set stylerlistsfes

`set` stylerlistsfes= { on | off}

This option is a preference that determines whether the Styler Formatting Elements (SFEs) configured for a stylesheet are displayed along with regular elements in the **Elements** list in Arbortext Styler. This preference can be set to `on` or `off`. A value of `on` displays SFEs in the list, while setting the preference to `off` removes them.

You can set this option as a preference from the **Tools ► Preferences** dialog box. Click the **Advanced** button and choose `stylerlistsfes`. Select `on` or `off` as the choice for the preference.

Setting this command to `on` is the equivalent of activating the **View ► Styler Formatting Elements** menu option in Arbortext Styler.

set stylerlistufes

`set` stylerlistufes= { on | off}

This option is a preference that determines whether the User Formatting Elements (SFEs) configured for a stylesheet are displayed along with regular elements in the **Elements** list in Arbortext Styler. This preference can be set to `on` or `off`. A value of `on` displays UFEs in the list, while setting the preference to `off` removes them.

You can set this option as a preference from the **Tools ► Preferences** dialog box. Click the **Advanced** button and choose `stylerlistufes`. Select `on` or `off` as the choice for the preference.

Setting this command to `on` is the equivalent of activating the **View ► User Formatting Elements** menu option in Arbortext Styler.

set stylernotbasefontcolor

`set` stylernotbasefontcolor= { *colorname* | *#rgbspec*}

This command determines the font color used to display the **Properties to edit** label in the Arbortext Styler window when the value of the field is any value except **Base (All Uses)**. The value is either a named color or an RGB specification preceded by `#`. The default is `red`.

Examples (the following are equivalent):

```
set stylernotbasefontcolor=red
set stylernotbasefontcolor=#ff0000
```

set stylerresolveconditions

`set` stylerresolveconditions= { on | off}

When set to `on` (the default), Arbortext Styler resolves conditions for the contexts selected in the Edit window.

This command corresponds to **Options ► Resolve Conditions** in Arbortext Styler.

set stylershowduplicatedefs

`set` stylershowduplicatedefs= { on | off}

This option is a preference that determines whether Arbortext Styler displays duplicate element definitions that are contained in both the main stylesheet and associated modules. This preference can be set to `on` or `off`. A value of `on` displays the duplicated definitions in the list, while setting the preference to `off` removes them.

You can set this option as a preference from the **Tools ► Preferences** dialog box. Click the **Advanced** button and choose `stylershowduplicatedefs`. Select `on` or `off` as the choice for the preference.

Setting this command to `on` is the equivalent of activating the **View ► Show Duplicate Definitions** menu option in Arbortext Styler.

set stylershowunstyled

`set` stylershowunstyled= { on | off}

When set to `on` (the default), Arbortext Styler highlights missing and unstyled elements in the Edit window, as well as in all Arbortext Styler previews. A missing element is any element that does not have a matching context in the stylesheet.

Missing and unstyled elements are highlighted by displaying the element start and end with a blue font color on a pink background. This option also generates new lines before and after these elements in Arbortext Styler previews.

This command corresponds to **Options ► Highlight Unstyled Elements** in Arbortext Styler.

set stylersyncelements

`set` stylersyncelements= { on | off}

When set to `on` (the default), placing the cursor inside an element in Arbortext Editor causes that element to be selected in the Arbortext Styler **Elements** list.

This command corresponds to the **Options ► Synchronize Elements with Editor** menu choice in Arbortext Styler.

set stylerunstyledfontcolor

`set` stylerunstyledfontcolor= { *colorname* | *#rgbspec*}

This command determines the font color of the element name in the unstyled indicator in any Arbortext Styler preview window when the element has a **Style** property of `unstyled`. The value is either a named color or an RGB specification preceded by `#`. The default is `blue`.

The unstyled indicator also has a background color that is configured using the set stylerunstyledfontshading on page 900 setting.

Examples (the following are equivalent):

```
set stylerunstyledfontcolor=red
set stylerunstyledfontcolor=#ff0000
```

set stylerunstyledfontshading

set stylerunstyledfontshading= { *colorname* | *#rgbspec*}

This command determines the font shading color of the element name in the unstyled indicator in any Arbortext Styler preview window when the element has a **Style** property of unstyled. The value is either a named color or an RGB specification preceded by #. The default is red.

The unstyled indicator also has a font color that is configured using the set stylerunstyledfontcolor on page 899 setting.

Examples (the following are equivalent):

```
set stylerunstyledfontshading=red
set stylerunstyledfontshading=#ff0000
```

set stylervalnestedpagesetsxslfo

set stylervalnestedpagesetsxslfo= { on | off}

The default value for this option is off. When set to on, Arbortext Styler requires strict XSL-FO compliance when validating page sets. Arbortext Styler is able to process nested XSL-FO page sets, so they are not reported as validation errors. However, if you plan to export your stylesheet as XSL-FO and use it in another application, nested page sets are not strictly XSL-FO compliant. If you enable this option, Arbortext Styler reports nested XSL-FO page sets as validation errors. This option is disabled if you are using FOSI as your Arbortext Styler print engine.

This command corresponds to the **Require XSL-FO Compliance** check box in the Arbortext Styler **Validate Page Sets** dialog box.

set stylerviewapproottags

set stylerviewapproottags= { on | off}

When set to on, tags in the PTC APP root template of a stylesheet are made available for edit in Arbortext Styler.

PTC APP root template tags display in the **Functions** list.

The default value is off.

This command corresponds to the **Tools ► Edit APP Root Template Source** menu option in Arbortext Styler.

set stylesheet

`set` stylesheet=*name*

This command specifies the stylesheet to be used in place of the default stylesheet for the Editor view.

name is the path and file name of a Arbortext Styler or FOSI stylesheet. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, then a `.fos` file.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

Examples

```
set stylesheet=memo2.style
set stylesheet=/mydocs/mem/memo2.fos
set stylesheet=http://www.some_site.com/examples/techman.xsl
```

set stylesheetassociations

`set` stylesheetassociations= { on | off}

When the `stylesheetassociations` option is set to `on`, which is the default, the user can specify stylesheet associations, and the stylesheet associations are stored as processing instructions in a document when it is saved. It might be necessary to turn this option off at some sites due to SGML or XML applications that are broken by the presence of stylesheet association processing instructions before the document type declaration.

When the option is set to `off`, there is no user interface for specifying stylesheet associations, and files that are saved are written without stylesheet association processing instructions, which also removes any pre-existing processing instructions.

If a stylesheet association is specified when Arbortext Editor is using Arbortext Publishing Engine for publishing, only the stylesheet name is saved to the document. If a stylesheet of the same name exists on the Arbortext PE server, it will appear in the stylesheet list of selections for the print and publish dialog boxes with the `(pe)` notation.

set tablecalscolnamerequired

`set` tablecalscolnamerequired= { on | off}

Controls whether new `entry` tags in a CALS table will have their `colname` attribute set. The default setting is `off`.

set tablecolumnaligncharacter

`set` tablecolumnaligncharacter= *char*

Controls the default alignment character (*char*) on the **Justification** tab of the **Table Properties** dialog box.

Examples

```
set tablecolumnalignchar="."
set tablecolumnalignchar="-"
```

set tablecolumnresizable

`set` tablecolumnresizable= { on | off}

Controls the ability to resize table columns. When set to `on` (the default), the user interface allows table column widths to change. When set to `off`, the following user interface changes occur:

- You cannot drag the column markers in the Column Tool.
- In the **Table Properties** dialog box, the **Column** tab is disabled.

set tabledefaultrulethickness

`set` tabledefaultrulethickness= *dimen*

The `set tabledefaultrulethickness` command specifies the rule thickness for tables in the current document. However, it has no impact on tables whose markup specifies a rule thickness.

This setting applies to all rules in existing CALS and Arbortext tables. It only applies to outer frame rules in HTML tables. Arbortext Editor also uses the `tabledefaultrulethickness` value as the default border width in the Insert Table dialog box for HTML tables.

Allowable units of measure are:

- `in` — inches (the default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points

- `pi` or `pc` — picas
- `px` — pixels

The default setting is `1pt` (one point).

Examples

```
set tabledefaultrulethickness=25pt
set tabledefaultrulethickness=1pc
```

You can view changes greater than 1pt in the Edit window.

set tableminimumemptyrowheight

`set` tableminimumemptyrowheight= *dimen*

This command determines the default published height for empty table rows. The default value is `15 pt`. Specify the numeric value and the unit of measure. Allowable units of measure are:

- `in` — inches (the default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pi` or `pc` — picas
- `px` — pixels

Examples

```
set tableminimumemptyrowheight=10cm
set tableminimumemptyrowheight=350pt
```

set tableminimumrowheight

`set` tableminimumrowheight= *dimen*

This command determines the default published height for table rows with content. The default value is `15 pt`. Specify the numeric value and the unit of measure. Allowable units of measure are:

- `in` — inches (the default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pi` or `pc` — picas
- `px` — pixels

Examples

```
set tableminimumrowheight=10cm
set tableminimumrowheight=350pt
```

set tablenewrowheightunit

`set` tablenewrowheightunit= { in | cm | mm | pt | pc | pi}

This command sets the units for assigning explicit heights to new rows. Available units are:

- `in` — inches (the default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pc` — picas
- `pi` — picas

Examples

```
set tablenewrowheightunit=in
set tablenewrowheightunit=pt
```

set tablerulers

`set` tablerulers= { on | off}

Controls the display of the Table Rulers. When set to `on`, the Row Ruler, Column Ruler and Table Icon appear in the edit view. When set to `off`, these interface items are hidden.

You can set the Window preference for displaying Table Rulers. Go to **Tools ▸ Preferences**, and, in the **Window** category, turn on **Table Rulers**.

You can also set Table Rulers display using the **View ▸ Tables ▸ Table Rulers** menu command.

set tablesavecolumnwidthunit

`set` tablesavecolumnwidthunit= { * | in | cm | mm | pt | pc | pi | off}

This command sets the units that Arbortext Editor will convert all column widths to when saving documents. This option does not apply to HTML tables. Available settings are:

- `*` — proportional
- `in` — inches
- `cm` — centimeters

- `mm` — millimeters
- `pt` — points
- `pc` — picas
- `pi` — picas
- `off` — disables column width conversion

set tabletagdisplay

`set` tabletagdisplay= { none | partial | full | inherit | cycle}

This command changes the tag display for table content where `none` does not display most tags or icons, `partial` displays small icons for most tags, `full` (the default) displays the fully spelled-out tags, `inherit` automatically follows the setting for `set tagdisplay`, and `cycle` moves among the four types of displays in the order full, partial, none, inherit.

Display characteristics for a particular tag pair in a particular mode are set with the `tag_display` on page 719 command.

set tabletags

`set` tabletags= { on | off}

Controls the display of tables. When set to `off`, the tags that define table structure are hidden (regardless of the current tag display) and a graphical representations of tables appear in the edit view. When set to `on`, this graphical structure is hidden.

If the current tag display (`set tagdisplay` command) is set to `full` or `partial`, you will see the tags that define a table's structure. If `set tabletags` is set to `on` and the current tag display is set to `none`, only the content of the table will be visible.

set tabletoolbarautohide

`set` tabletoolbarautohide= { on | off}

When set to `on`, `set tabletoolbarautohide` automatically hides the **Table** tool bar whenever the cursor is outside a table. This setting corresponds to the **Table Toolbar Only When Table is Active** check box of the **Window** tab of the **Tools ► Preferences** dialog box. The default is `off`.

set tableuiextensions

`set` tableuiextensions= { on | off}

Several forms of table formatting are saved in the document as processing instructions. These processing instructions are not necessarily portable to other authoring and publishing tools. The `tableuiextensions` option controls whether the user interface will allow an author to apply table formatting that is saved as a processing instruction.

When this option is set to `on` (the default), all table formatting is available. When set to `off`, several pieces of the user interface become disabled:

- The **Style**, **Color**, and **Width** options are not available on the **Modify Borders** dialog box.
- The **Modify Font** dialog box is not available from within the **Cell** tab of the **Table Properties** dialog box.
- For OASIS Exchange tables, the **Table** tab of the **Table Properties** dialog box is not available.
- The **Published Output** option does not appear on the **Row** tab of the **Table Properties** dialog box.
- For OASIS Exchange and Arbortext tables, you cannot specify a **Shading** option.
- For OASIS Exchange tables, you cannot modify the **Row Height** on the **Row** tab of the **Table Properties** dialog box. Similarly, you cannot drag the row height markers on the Row Tool.
- For HTML tables, you cannot modify individual cell rules in the **Modify Borders** dialog box.

Note that this command controls the ability to make future changes to table formatting. It will not eliminate existing processing instructions previously added to support formatting changes.

set tablewidth

`set` tablewidth= *dimen*

This command determines the default screen display width for tables in the Edit window. The default value is `100%`. Specify the numeric valued and the unit of measure. Allowable units of measure are:

- `%` — percent
- `in` — inches (the default)
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points

- `pi` or `pc` — picas
- `px` — pixels

The *APTTBLWIDTH* environment variable can set the default. The `set units` on page 912 option can set the unit of measurement.

Examples

```
set tablewidth=10cm
set tablewidth=350pt
```

set tablewriteemptycellmarkup

`set` tablewriteemptycellmarkup= { on | off}

Arbortext Editor creates table cells which contain `<entry> </entry>` tag pairs. However, Arbortext Editor can edit tables with tables cells not containing `<entry> </entry>` tag pairs created by other applications.

When `tablewriteemptycellmarkup` is set to `off`, Arbortext Editor omits the `<entry> </entry>` tag pairs when saving such a document. Certain other circumstances, described in the *Handling empty table cells* help topic, can cause Arbortext Editor to include the tags when saving the document.

When `tablewriteemptycellmarkup` is set to `on`, Arbortext Editor saves the empty `entry` elements in the document.

The default setting for `tablewriteemptycellmarkup` is `off`.

set tagdisplay

`set` tagdisplay= { none | partial | full | cycle}

This command changes the tag display where `none` (the default) does not display most tags or icons, `partial` displays small icons for most tags, `full` displays the fully spelled-out tags, and `cycle` moves among the three types of displays in the order full, partial, none

Display characteristics for a particular tag pair in a particular mode are set with the `tag_display` on page 719 command.

set tagfontcolor

`set` tagfontcolor= { *colorname* | *#rgbspec*}

This command determines the font color used to display markup icons in the Edit window. The value is either a named color or an RGB specification preceded by `#`. The default is `brown`.

You can determine whether Arbortext Editor ignores this setting by using the `set usecolorsettings` on page 912 option.

Examples

```
set tagfontcolor=red
set tagfontcolor=#ff0000
```

set tagfontpercent

`set` tagfontpercent= *n*

This command increases or decreases the size of all tags to the percentage specified. The default value is system dependent. Fonts cannot be displayed at less than 40% of their point size.

n is a number, excluding a percent (`%`) symbol.

set tagscan

`set` tagscan= { on | off}

The `tagscan` option allows you to use search strings with the `find` on page 646, `caret` on page 621, and `window` on page 727 commands without specifying the `-t` modifier. `off` is the default.

set tagtemplatepath

`set` tagtemplatepath=*d1* [:*dn*]

This command specifies a list of directories to search for tag templates. Specify each directory separated by a semicolon. A list of multiple directories must be enclosed in quotation marks. The default path is the path specified by *APTTAGTPLDIR* environment variable, if it exists. If that environment variable is not set, the directory specified by the *$home* ACL variable is used.

If there is an *Arbortext-path*`\custom\tagtemplates` subdirectory at startup, the `custom\tagtemplates` path is automatically prepended to the tag template path. Putting your tag templates in the `custom\tagtemplates` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

You can 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.

You can also specify alternate or additional locations for your tag templates directory via the document type's DCF file.. Refer to *Specifying an alternate location for a tag templates directory* in the *Document Types Guide* for information.

You can use the `append_tagtemplate_path` function on page 224 to update the tag template path.

set textentityfontcolor

`set` textentityfontcolor= { inherit | *colorname* | *#rgbspec*}

This command determines the font color used to display text entities in the Edit window. If `inherit` is set, the text entities are displayed the same as the text of the Edit window document. The value is either `inherit`, or a named color, or an RGB specification preceded by `#`. The default is `inherit`.

You can determine whether Arbortext Editor ignores this setting by using the `set usecolorsettings` on page 912 option.

Examples

```
set textentityfontcolor=inherit
set textentityfontcolor=red
set textentityfontcolor=#ff0000
```

set textentitymarkers

`set` textentitymarkers= { none | full | partial | default}

This command determines the display of text entity markup icons when text entities are displayed in the Edit window document. When set to any of the following, the setting determines the display regardless of the Edit window document tag display setting:

- `none` — No tags or markers are displayed.
- `full` — All markup tags and markers are displayed.
- `partial` — Only the file entity markers around the file entity are displayed.

When set to `default`, text entity markup takes on the same setting as the Edit window document tag display. The default is `none`.

set toolbar

`set` toolbar= { on | off}

The `toolbar` option turns `on` or `off` the display of the **Edit** (toolbar1), **Markup** (toolbar2), **Table** (toolbar3), and **Application** (toolbar4) tool bars. It is an easy way to show or hide all tool bars without changing the individual setting for each tool bar. The default is `on`.

Note

After you turn set `toolbar` to `on`, the table tool bar will display regardless of the current setting of `tabletoolbarautohide`. The next time you click within a table, the auto-hide behavior will resume its normal operation.

set toolbar1

`set` toolbar1= { on | off}

The `toolbar1` option turns `on` or `off` the **Edit** tool bar. The default is on.

set toolbar2

`set` toolbar2= { on | off}

The `toolbar2` option turns `on` or `off` the **Markup** tool bar. The default is `on`.

set toolbar3

`set` toolbar3= { on | off}

The `toolbar3` option turns `on` or `off` the **Table** tool bar. The default is on.

set toolbar4

`set` toolbar4= { on | off}

The `toolbar4` option turns `on` or `off` the Application tool bar.

set toolbar5

`set` toolbar5= { on | off}

The `toolbar5` option turns `on` or `off` toolbar5. Use this command if you have created toolbar5 using XUI.

set traceback

`set` traceback=*n*

The `set traceback` command allows you to set the number of levels displayed in the function call traceback if a user-defined ACL function aborts with a runtime error. The default is `10`. If you set `traceback` to `0` (zero), you disable the function call traceback.

Examples

```
set traceback=20
set traceback=6
set traceback=0
```

set trackcontext

`set` trackcontext= { on | off}

This command enables or disables the display of the element structure containing the cursor in the status bar (for example, memo body para | para). The default is `off`.

Note

If you have applied an alias map to the document and `set trackcontext = on`, aliases will display in the status bar for elements that have been assigned aliases.

set undolimit

`set` undolimit= *n*

This command specifies the maximum memory, in bytes, to use for `undo` buffer space for each open document. The default setting is 500 KB (or 512,000 bytes) of memory. The preference can be adjusted to a number of between 1 and 66536 to allow for more memory to be made available for the `undo`. If set to `0`, there is no limit other than available virtual memory. Note that if you set this number too low, you will inhibit `undo` capabilities. Setting the preference to an excessively high value could affect performance.

The undo limit is based on volume of data deleted. Arbortext Editor holds the deleted data in memory.

The undo buffer is allocated for each document. When a document is closed, its undo history will be discarded and the memory released.

Note

A change to this option does not take effect until the next document is opened. Setting this option to a low limit may prevent the `undo` command from restoring the last change.

Examples

```
set undolimit=50000
set undolimit=0
```

set units

`set` units= { in | cm | mm | pt | pi | px}

This option specifies the units of measurement used for table width display if the `set tablewidth` on page 906 option is not set to a `%` value. The default is `in`. The units of measurement are:

- `in` — inches
- `cm` — centimeters
- `mm` — millimeters
- `pt` — points
- `pi` — picas
- `px` — pixels

set updaterecentdocuments

`set` updaterecentdocuments= { on | off}

When the this option is `on` (the default), a file added to the recently used list of files on the **File** menu is also added to the Windows **Start ► My Recent Documents** list.

set usecolorsettings

`set` usecolorsettings= { on | off}

This option determines whether to use the `set fileentityfontcolor` on page 797, `set textentityfontcolor` on page 909, `set gentextfontcolor` on page 818, and `set tagfontcolor` on page 907 options. When set to `off`, those options are ignored, and the contents of entities, generated text, and tags are drawn with the same font color as surrounding text.

This option has a view scope and defaults to `on`.

set useepic43keymappings

`set` useepic43keymappings= { on | off}

The default keyboard mappings used in Epic Editor 4.4 and beyond are different than those used in the 4.3 release. If you prefer the 4.3 version of keyboard mappings, you can turn this option on to restore them.

This option has a global scope and defaults to `off`. This option takes effect the next time you start Arbortext Editor.

The following table shows the keyboard shortcut for each Arbortext macro (for executing Arbortext Editor actions and menu selections), with the version 4.3 and updated keyboard mappings. If a column is blank it means that there is no shortcut for that command in the corresponding release.

Command assignments

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
AccentAcute	**Alt+'**	**Alt+'**
AccentBarBelow	**Alt+_**	**Alt+_**
AccentBreve	**Alt+[**	**Alt+[**
AccentCaron	**Alt+<**	**Alt+<**
AccentCedilla	**Alt+;**	**Alt+;**
AccentCircumflex	**Alt+^**	**Alt+^**
AccentDot	**Alt+.**	**Alt+.**
AccentDotBelow	**Alt+,**	**Alt+,**
AccentDoubleAcute	**Alt+#**	**Alt+#**
AccentGrave	**Alt+`**	**Alt+`**
AccentMacron	**Alt+-**	**Alt+-**
AccentOgonek	**Alt+(**	**Alt+(**
AccentRing	**Alt+@**	**Alt+@**
AccentTilde	**Alt+~**	**Alt+~**
AccentUmlaut	**Alt+"**	**Alt+"**
AssignStyle	**Ctrl+Shift+H**	
BackSpace	**BackSpace**	**BackSpace**
Bold		**Ctrl+B**
BookmarkTool	**Ctrl+B**	
Bookmarks	**Ctrl+Shift+B**	**Ctrl+Shift+F5**
Cancel	**Escape**	**Escape**
ChangeCaseToggle		**Shift+F3**
CharLeft	**LeftArrow**	**LeftArrow**

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
CharLeftExtend	**Shift+LeftArrow**	**Shift+LeftArrow**
CharRight	**RightArrow**	**RightArrow**
CharRightExtend	**Shift+RightArrow**	**Shift+RightArrow**
CheckCompleteness	**Shift+Alt+C**	
ClearSelection	**Shift+Alt+F5**	**Shift+Alt+F8**
ClickCollapseExpand	**Shift+M3**	**Shift+M3**
ClickCollapseExpandDivision	**Ctrl+M2**	**Ctrl+M2**
ClickContextMenu	**M2**	**M2**
ClickControlShift	**Ctrl+Shift+M1**	**Ctrl+Shift+M1**
ClickExtendUp	**Shift+Up+M1**	**Shift+Up+M1**
CloseOrExit	**Alt+F4**	**Alt+F4**
CollapseDivision		**Shift+Alt+Num_-**
CollapseToLevel0	**Ctrl+Shift+0**	**Ctrl+Shift+0**
CollapseToLevel1	**Ctrl+Shift+1**	**Ctrl+Shift+1**
CollapseToLevel2	**Ctrl+Shift+2**	**Ctrl+Shift+2**
CollapseToLevel3	**Ctrl+Shift+3**	**Ctrl+Shift+3**
CollapseToLevel4	**Ctrl+Shift+4**	**Ctrl+Shift+4**
CollapseToLevel5	**Ctrl+Shift+5**	**Ctrl+Shift+5**
CollapseToLevel6	**Ctrl+Shift+6**	**Ctrl+Shift+6**
CollapseToLevel7	**Ctrl+Shift+7**	**Ctrl+Shift+7**
CollapseToLevel8	**Ctrl+Shift+8**	**Ctrl+Shift+8**
CollapseToLevel9	**Ctrl+Shift+9**	**Ctrl+Shift+9**
CollapseTool	**Ctrl+.**	**Ctrl+.**
ColumnBreak		**Ctrl+Shift+Enter**
ComposeTagBegin	**Ctrl+'**	**Ctrl+'**
ContextMenu	**Shift+F10**	**Menu**
ControlClickUp	**Ctrl+Up+M1**	**Ctrl+Up+M1**
CopyText		**Shift+F2**
CreateTextEntity		**Alt+F3**
CursorAttributes	**Shift+Alt+A**	**Shift+Alt+A**
DeleteAllAttributes	**Ctrl+Shift+A**	**Ctrl+Shift+D**
DeleteCharacter	**Ctrl+H**	
DeleteMarkup	**Ctrl+D**	**Ctrl+Shift+X**
DeleteToEnd	**Alt+Delete**	**Alt+Delete**
DeleteWord	**Ctrl+Delete**	**Ctrl+Delete**
DeleteWordLeft	**Ctrl+BackSpace**	**Ctrl+BackSpace**

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
EditCopy	Ctrl+C	Ctrl+C
EditCopyAppend	Ctrl+Shift+C	
EditCut	Ctrl+X	Ctrl+X
EditCutAppend	Ctrl+Shift+X	
EditCutSpike		Ctrl+F3
EditGoTo		Ctrl+G
EditPaste	Ctrl+V	Ctrl+V
EditRedo	Ctrl+Shift+Z	Ctrl+Shift+Z
EditRedoOrRepeat		Ctrl+Y
EditUndo	Ctrl+Z	Ctrl+Z
ElementEnd	Alt+RightArrow	Alt+RightArrow
ElementEndExtend	Shift+Alt+RightArrow	Shift+Alt+RightArrow
ElementStart	Alt+LeftArrow	Alt+LeftArrow
ElementStartExtend	Shift+Alt+LeftArrow	Shift+Alt+LeftArrow
EndOfDocExtend	Ctrl+Shift+End	Ctrl+Shift+End
EndOfDocument	Ctrl+End	Ctrl+End
EndOfLine	End	End
EndOfLineExtend	Shift+End	Shift+End
EndOfWindow	Ctrl+PageDown	Ctrl+PageDown
EndOfWindowExtend	Ctrl+Shift+PageDown	Ctrl+Shift+PageDown
Enter	Enter	Enter
ExpandAll	Ctrl+0	Ctrl+0
ExpandDivision		Shift+Alt+Num_+
ExpandOneLevel	Ctrl++	Ctrl++
ExtendSelection	F8	F8
FileClose		Ctrl+W
FileEntityTool	Ctrl+K	
FileNew	Ctrl+N	Ctrl+N
FileNewDialog		Ctrl+Shift+N
FileOpen	Ctrl+O	Ctrl+O
FilePrint	Ctrl+P	Ctrl+P
FilePrintPreview	Shift+Alt+P	Ctrl+Alt+P
FileSave	Ctrl+S	Ctrl+S
FileSaveAll	Ctrl+Shift+S	Ctrl+Shift+S
FileSaveAs		F12
FindNext	Ctrl+Shift+F	Ctrl+Alt+Y
FindPrevious		Ctrl+Alt+F
FindReplace	Ctrl+F	Ctrl+F

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
FindSelection	Shift+Alt+F	Shift+Alt+F
GoToQuickmark	Ctrl+Q	Ctrl+Q
HelpAtMouse	F1	Shift+F1
HelpTopics		F1
InsertComment		Ctrl+Alt+M
InsertCopyright		Ctrl+Alt+C
InsertEquation	Shift+Alt+Q	Shift+Alt+Q
InsertFileEntity	Ctrl+Shift+K	
InsertGraphic	Shift+Alt+G	Shift+Alt+G
InsertLink		Ctrl+K
InsertMarkup	Ctrl+Shift+I	Ctrl+Shift+M
InsertMarkupTool	Ctrl+I	Ctrl+M
InsertMdash	Ctrl+Shift+_	Ctrl+Shift+_
InsertNbsp	Ctrl+Space	Ctrl+Shift+Space
InsertNdash	Ctrl+-	Ctrl+-
InsertQuickmark	Ctrl+Shift+Q	Ctrl+Shift+Q
InsertRegistered		Ctrl+Alt+R
InsertSpace	Shift+Space	Shift+Space
InsertSymbol	Ctrl+Shift+R	Ctrl+Shift+R
InsertTable	Shift+Alt+T	Shift+Alt+T
InsertTextEntity	Ctrl+Shift+Y	
InsertTrademark		Ctrl+Alt+T
Italic		Ctrl+I
LeftClick	M1	M1
LeftClickExtend	Shift+M1	Shift+M1
LeftClickUp	Up+M1	Up+M1
LeftControlClick	Ctrl+M1	Ctrl+M1
LeftDoubleClick	Dbl+M1	Dbl+M1
LeftTripleClick	Trpl+M1	Trpl+M1
LineBreak	Shift+Enter	Shift+Enter
LineDown	DownArrow	DownArrow
LineDownExtend	Shift+DownArrow	Shift+DownArrow
LineUp	UpArrow	UpArrow
LineUpExtend	Shift+UpArrow	Shift+UpArrow
LinkActivate	Ctrl+Shift+O	Ctrl+Shift+K
LinkGoBack	Ctrl+Shift+P	Ctrl+Alt+Z
LowerCase	Shift+F7	
ModifyAttributes	Ctrl+A	Ctrl+D

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
ModifyElements	Ctrl+Shift+N	
ModifyStyle	Ctrl+Shift+M	
MoveText		F2
NextMisspelling		Alt+F7
NextPane	Alt+F6	Alt+F6
NextWindow	Ctrl+F6	Ctrl+F6
OtherPane	F6	F6
Overstrike	Insert	Insert
PageBreak	Ctrl+Enter	Ctrl+Enter
PageDown	PageDown	PageDown
PageDownExtend	Shift+PageDown	Shift+PageDown
PageUp	PageUp	PageUp
PageUpExtend	Shift+PageUp	Shift+PageUp
ParaDown	Ctrl+DownArrow	Ctrl+DownArrow
ParaDownExtend	Ctrl+Shift+DownArrow	Ctrl+Shift+DownArrow
ParaUp	Ctrl+UpArrow	Ctrl+UpArrow
ParaUpExtend	Ctrl+Shift+UpArrow	Ctrl+Shift+UpArrow
PasteInsertAfter		Ctrl+Shift+V
PreviousPane	Shift+Alt+F6	Shift+Alt+F6
PreviousWindow	Ctrl+Shift+F6	Ctrl+Shift+F6
ProperCase	Ctrl+F7	
QuickTags	KeyPad_Enter	Keypad_Enter
RepeatCommand	Shift+Alt+R	Shift+Alt+R
ReplaceAgain		Ctrl+H
ReplaceAll		Ctrl+Shift+H
Scroll5LinesDown	Shift+Alt+DownArrow	Shift+Alt+DownArrow
Scroll5LinesUp	Shift+Alt+UpArrow	Shift+Alt+UpArrow
ScrollLineDown	Alt+DownArrow	Alt+DownArrow
ScrollLineUp	Alt+UpArrow	Alt+UpArrow
SelectAll		Ctrl+A
SelectElement	Ctrl+Shift+E	
SelectElementContent	Ctrl+E	Ctrl+E
SentenceCase	Ctrl+Shift+F9	
ShowGentext	Ctrl+G	Alt+Ctrl+G
ShrinkSelection		Shift+F8
Spelling	Alt+$	F7
StartOfDocExtend	Ctrl+Shift+Home	Ctrl+Shift+Home
StartOfDocument	Ctrl+Home	Ctrl+Home

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
StartOfLine	Home	Home
StartOfLineExtend	Shift+Home	Shift+Home
StartOfWindow	Ctrl+PageUp	Ctrl+PageUp
StartOfWindowExtend	Ctrl+Shift+PageUp	Ctrl+Shift+PageUp
SymbolTool	Ctrl+R	Ctrl+R
SyncOtherPane	Ctrl+=	Ctrl+=
TabLeft	Shift+Tab	Shift+Tab
TabRight	Tab	Tab
TableDeleteColumn	Ctrl+F3	Ctrl+F11
TableDeleteRow	Ctrl+F2	F11
TableInsertColumn	F3	Ctrl+F9
TableInsertColumn Right	Shift+F3	Ctrl+Shift+F9
TableInsertRow	F2	F9
TableInsertRowBelow	Shift+F2	Shift+F9
TableSelectCell	F5	Alt+F9
TableSelectColumn	Ctrl+F5	Ctrl+Alt+F9
TableSelectRow	Shift+F5	Shift+Alt+F9
TableSelectTable	Ctrl+Shift+F5	Alt+Num_5
TableSpanCells	F4	Alt+F11
TableUnspanCells	Shift+F4	Shift+Alt+F11
TextEntityTool	Ctrl+Y	Ctrl+Alt+V
Thesaurus		Shift+F7
ToggleAutoSyncPanes	Ctrl+<	
ToggleFrameSplitBelow		Ctrl+Alt+S
ToggleGentext	Ctrl+Shift+G	
ToggleTagDisplay	Ctrl+Shift+L	Ctrl+Shift+L
ToggleUpperCase		Ctrl+Shift+A
TrackChanges		Ctrl+Shift+E
Transpose	Ctrl+T	Ctrl+T
TransposeWords	Ctrl+Shift+T	Ctrl+Shift+T
Underline		Ctrl+U
UpperCase	F7	
ViewDocmap		Ctrl+Alt+O
ViewNormal		Ctrl+Alt+N
WindowMaximize		Alt+F10
WindowMove		Ctrl+F7
WindowRemoveSplit		Shift+Alt+C

Command assignments (continued)

Command	Epic Editor 4.3	Epic Editor 4.4 (and later)
WindowRestore		**Alt+F5**
WindowSize		**Ctrl+F8**
WordCount		**Ctrl+Shift+G**
WordLeft	**Ctrl+LeftArrow**	**Ctrl+LeftArrow**
WordLeftExtend	**Ctrl+Shift+LeftArrow**	**Ctrl+Shift+LeftArrow**
WordRight	**Ctrl+RightArrow**	**Ctrl+RightArrow**
WordRightExtend	**Ctrl+Shift+RightArrow**	**Ctrl+Shift+RightArrow**
ZoomDown	**Ctrl+Shift+F8**	**Ctrl+<**
ZoomUp	**Ctrl+F8**	**Ctrl+>**

set user

`set` user=*user*

This option specifies the user ID for the current user. The default value is defined by the system, and may be user-specific if the operating system requires every user to log-in.

If the system provides no default and the option is not set then the first time the *user* value is needed, a popup will appear to request the *user* and *fullname* values. If the popup is dismissed (for example, when running under Arbortext Publishing Engine all pop-ups are dismissed automatically) then a unique session id will be used for user.

The value of *user* is associated with tracked changes and is put into each change record as the record is created.

For example, if the user's name is Edgar Allan Poe, the user id could be:

```
set user=eap
```

set usercolor

`set` usercolor= { *colorname* | *#rgbspec*}

This command allows you to set the user color. The value is either a named color or an RGB specification preceded by `#`. By default, no color is selected. This `set` option corresponds to the **User Color** preference in **Tools ► Preferences ► User Information**.

This color can be used to identify the changes you make when change tracking is active. This color identifies you in the Edit window, in print and in published output. If you do not choose a user color, a color will still be assigned to you to

identify you in a document. If this setting conflicts with another user's color, you may be assigned a different color for that document only. This value will not be changed.

Examples (the following are equivalent):

```
set usercolor=red
set usercolor=#ff0000
```

set userdictpath

`set` userdictpath=*d1* [:*dn*]

This command searches a list of directories, separated by semicolons, for user-defined dictionaries that are used by the Spell Checker. Specify the directories containing the dictionaries; don't specify individual dictionary names. The default search path is *`Arbortext-path`*`\lib\proximity\userdict`. You can specify the `%D` or `%H` symbolic parameters in the path list in the path, `%D` specifies the default search path.

If there is an *`Arbortext-path`*`\custom\dictionaries` subdirectory at startup, the `custom\dictionaries` path is automatically prepended to the dictionary path. Putting your custom dictionaries in the `custom\dictionaries` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

You can use the `append_userdict_path` on page 223 function to update the tag template path. You can also specify the user dictionary path in **Tools ► Preferences ► Spelling** dialog box.

Example

```
set userdictpath=/users/pat:/company/dicts
set userdictpath="c:\Program Files\Arbortext\editor\dict;c:\Arbortext\mydict"
```

set userinput

`set` userinput= { on | off}

Setting `userinput` to `off` blocks user input to the Edit window. If `set userinput` is set to `on` (the default), input is not blocked.

> **Caution**
>
> Use this command with extreme care. An application should only block input if it has an alternative method of communicating with Arbortext Editor (for example, if Arbortext Editor is acting as a server).

set userules

`set` userules= { on | off}

When on, values of `usetexts` with non-zero `userules` are inserted into "preliminary userule oids" and are processed by `userule` processing. When off, `usetexts` with non-zero `userules` are completely ignored. This saves processing time and memory use. Do not use the `off` setting when creating final output.

The default is on.

Note

Be careful when using the `off` setting with an unfamiliar FOSI. It is possible that non-zero `userules` are used for effects other than indexes in the FOSI.

set usexsdasdtd

`set` usexsdasdtd= { on | off}

This set option provides a means to migrate a document instance from a DTD to an XML Schema.

If set to `on`, an instance with a DOCTYPE declaration resolving to a `.dtd` file will instead be parsed against the XML Schema if there is a `.xsd` file with the same base name as the `.dtd` in the document type directory. In this case, the ISO standard character entities will automatically be converted to the equivalent Unicode characters. Character entities that cannot be converted will be flagged as undefined entities because an XML schema cannot declare entities.

The default is `off`. This option has no effect for SGML document types.

set validatenamespaces

`set` validatenamespaces= { on | off}

When set to `on`, this option specifies that Arbortext Editor will validate foreign namespaces used in a document and therefore disallow any elements not declared in the document type. When the option is `off` (the default), any element name with a prefix associated with a foreign namespace is considered valid.

set validationmode

`set` validationmode= { all | dtd | schema}

Specifies how documents in the current session are to be validated.

- `all` — The default. Validates documents against their declared DTD or schema. Documents with both a DTD and schema declared will be validated against each document type. The document will be validated against the schema before the DTD.
- `dtd` — Validate documents against their declared DTD only.
- `schema` — Validate documents against their declared schema only.

set vertspacefulltags

`set` vertspacefulltags= { on | off}

This command enables or disables the display of vertical space due to FOSI specified pre-space and post-space when the tag display is **full**. The default is `on`.

set vertspacemax

`set` vertspacemax= *dimen*

This command determines the maximum space that is displayed before or after a vertical element when the display of vertical space is enabled. Allowable units of measure are: `pc, pt, cm, mm, and in`. The default is `14.46pt` (2 inches).

Example

```
set vertspacemax=1.5in
```

set vertspacemin

`set` vertspacemin= *dimen*

This command determines the minimum space that is displayed before or after a vertical element when the display of vertical space is enabled. Allowable units of measure are: `pc, pt, cm, mm, and in`. The default is `0pt`.

Example

```
set vertspacemin=10pt
```

set vertspacenotags

`set` vertspacenotags= { on | off}

This command enables or disables the display of vertical space when the tag display is set to **none**. The default is `on`.

set vertspacepartialtags

`set` vertspacepartialtags= { on | off}

This command enables or disables the display of vertical space when the tag display is set to **partial**. The default is `on`.

set vertspacepercent

`set` vertspacepercent= *n*

This command determines the percentage of the vertical space shown in the `Edit` window. The default is `100`.

n is a number, excluding a percent (`%`) symbol.

set view

`set` view= { edit | docmap | column}

This command determines the view displayed in the Arbortext Editor window. When set to `edit`, the Edit view is displayed. When set to `docmap`, the Document Map is displayed. When set to `column`, the Column view is displayed.

This option has local view scope and is not saved to the user preferences file.

set viewchangetracking

`set` viewchangetracking= { original | changesapplied | changeshighlighted}

This option specifies the type of change tracking markup to display. It applies to the current Document Map and Edit view pair. The default setting is `changesapplied`.

- *original* — Shows document as if all pending change records have been rejected.
- *changesapplied* — Shows document as if all pending change records have been accepted.
- *changeshighlighted* — All pending change records are visible.

When this command is set to `original`, the view is read-only (and will not be affected by changing `rochange`).

Example

To view the document without any pending changes displayed:

```
set viewchangetracking=original
```

set viewmode

`set` viewmode= { edit | docmap | column}

This command determines the view displayed in the Arbortext Editor window. When set to `edit`, only the Edit view is displayed. When set to `docmap`, the Document Map is displayed in a split view with the Edit view. When set to `column`, the Column view is displayed in a split view with the Edit view. The position of the split between the two views is determined by the `set docmapside` option. The percentage of the window allocated to the Document Map or Column view is determined by the `set docmappercent` option.

This option has local window scope and is saved to the user preferences file.

set webstylesheet

`set` webstylesheet= { *name* | none}

This command specifies the stylesheet to be used as the default for the **Publish ▸ For Web** option on the **File** menu.

name is the path and file name of a `.style` or `.xsl` stylesheet file. If you don't specify an extension for the stylesheet, Arbortext Editor first looks for a `.style` file, and then a `.xsl` file.

The stylesheet ID processing instruction must specify `web` as the publishing type.

If you specify only the stylesheet's base name, Arbortext Editor first looks for the file in the document directory, and then looks in the document type directory. If you specify a relative path (other than one that just gives the base name), Arbortext Editor looks for the file using the given path relative to the current working directory.

If you are using Arbortext Publishing Engine for publishing, it must be able to locate the stylesheet. If a document references a stylesheet using a URL, that stylesheet can be found by both Arbortext Editor and Arbortext Publishing Engine. Stylesheets that are available on the Arbortext PE server appear in the Stylesheet selection fields preceded by the notation `(pe)`.

When you are using Arbortext Publishing Engine, be aware of the following when setting a value for this option:

- If you try to set the location to a local path for any type of published output, you will get an error that the stylesheet is not the name of a stylesheet on the server. The following example produces an error if Arbortext Publishing Engine publishing is enabled:
  ```
  set webstylesheet=D:\ArbortextUser\axdocbook.style
  ```
- If you try to set the location to a stylesheet file name only (no path), Arbortext Publishing Engine looks for a stylesheet with a matching name. The following

example would succeed if a stylesheet of the same name exists in a location where the Arbortext PE server looks for stylesheets.

```
set webstylesheet=axdocbook.style
```

If `webstylesheet` is set to `none` or is not set, Arbortext Editor and Arbortext Publishing Engine use the Editor/Default stylesheet.

Examples

```
set webstylesheet=memo2
set webstylesheet=~/mydocs/memo/memo2.style
set webstylesheet=http://www.some_site.com/examples/techman_web.xsl
```

set webzonepolicy

`set` webzonepolicy= { on | off}

Determines whether the Microsoft URL security zone policy is used to determine whether a link using the `arbortext-editor:` protocol is allowed from a web browser. If `webzonepolicy` is set to `on` (the default) and a link using the `arbortext-editor:` protocol is invoked from a web browser, Arbortext Editor first determines the zone of the encoded document 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. If `webzonepolicy` is set to `off`, then no links using the `arbortext-editor:` protocol are allowed at all and a similar message is displayed.

set windows

`set` windows= { open | closed}

The `windows` option closes all open Arbortext Editor windows when set to `closed`. Setting `windows` to `open` (the default) opens them up again.

Note

The `windows` option is intended for use by a client attached using the socket API. If you issue this command from the Arbortext Editor command line, you will be unable to re-open the Arbortext Editor windows.

set windowsscriptdebugger

`set` windowsscriptdebugger= { on | off}

The `windowsscriptdebugger` option controls whether any installed Windows script debugger can be launched.

If `windowsscriptdebugger` is set to `on`, any system-installed Windows script debugger will be launched automatically when the active script engine encounters one of the following conditions:

- A run-time error. (Script syntax errors will not launch the debugger.)
- A `Stop` statement in a VBScript.
- A `debugger` statement in a JScript.

Do not set this option to `on` if no debugger is installed.

If `windowsscriptdebugger` is set to `off`, no script debugger will be launched. Runtime errors will be reported in message boxes.

The default value of the `windowsscriptdebugger` option is `off`.

set wordincludechars

`set` wordincludechars= *chars*

Specifies a list of punctuation characters that Arbortext Editor should considered as word-constituent characters for selection. The default value is the null string so underscores and dashes are not considered part of a word selection by default.

Example:

```
set wordincludechars='-_()$'
```

set wordscan

`set` wordscan= { on | off}

When this command is set to `on`, the `find` and `substitute` commands consider whole words only when matching text. For example, the search string does not find then or there. The default is `off` .

> **Note**
>
> This option has no effect if regular expressions are used. The regular expression meta-character `\b` can be used to match a word boundary instead, for example, `\bthe\b`.

set wrapprompt

`set` wrapprompt= { on | off}

When this command set to `on`, a prompt message is issued by the `find` or `substitute` commands when they reach the end (or beginning) of the document. The default is `off`.

set wrapscan

`set` wrapscan= { on | off}

When the `wrapscan` command is set to `on` (the default), the `find` and `substitute` commands revert to the beginning of the document to continue a search (or to the end for a reverse search).

set writeabsolutesysid

`set` writeabsolutesysid= { on | off}

This option determines whether the `save` and `write` commands will update the system identifier on the DOCTYPE declaration when writing XML documents. If the option is `off` (the default), the system ID will not be changed if it is not null. If the original system ID is null, Arbortext Editor writes the base name of the DTD corresponding to the document type. If the option is `on`, then Arbortext Editor writes an absolute URI for the system ID corresponding to the document type used to load the document (or to the `-public` option if given). If the `-sysid` option is used on the `write` command, then that value is used regardless of the `writeabsolutesysid` option setting.

The `set writeabsolutesysid` command also affects the `href` written for stylesheet associations. If this options is off, the `hrefs` of existing stylesheet associations are written unchanged, and for new stylesheet associations where the

stylesheet is in the same directory as the document or the DTD, a relative URI is written for the `href`. When this option is on, relative URIs are converted to absolute `file://` URI specifications.

set writeaticomment

`set` writeaticomment= { on | off}

This option instructs Arbortext Editor to write a Arbortext identifying comment as the second line of any saved SGML or XML file. The comment includes the `<!DOCTYPE...>` declaration for the document type.

If this option is set to `off`, the comment is suppressed.

The default setting is `on`.

set writechangetracking

`set` writechangetracking= { original | changesapplied | changeshighlighted}

Specifies the write command output:

- `original` — Show the document as if all pending change records have been rejected.
- `changesapplied` — Shows the document as if all pending change records have been accepted.
- `changeshighlighted` — Shows all pending change records.

The default is `changeshighlighted`.

Note

The `write` on page 728 command can override this set command option.

Example

To set the write output to show the document as if all pending changes have been rejected:

```
set writechangetracking=original
```

set writecheck

`set` writecheck= { on | off}

When this command is set to `on` (the default), a warning is issued if you try to use the `write` command to write over an existing file.

set writeentdecls

`set` writeentdecls= { rootallchildall | rootallchildref | rootallchildnone | roottreerefchildref | roottreerefchildnone | rootrefchildref}

This option controls which entity declarations are written in the subsets of the root document and its children when these are written by either the `save` or `write` commands. (Children include file entities and embedded objects. When the documentation below refers to file entities, the discussion applies to embedded objects as well.)

Generally, this option should be used consistently within an application. It may not be desirable for individuals to have their own settings, or for this to be frequently changed, though some settings have value when used temporarily for clean up purposes.

If it is desired that an entity declaration in the internal subset be preserved when there are no references to that entity anywhere in the document instance, including the internal subset itself (for instance, a parameter entity only referenced from the external subset of the DTD), then `writeentdecls` must be set to one of the three `rootall` values.

The following table summarizes the choices. More detailed explanations follow the table.

`rootallchildall`	All entity declarations of the document tree should be written in the root document's subset, and in the subset of all children of the document tree.
`rootallchildref`	All entity declarations of the document tree should be written in the root document's subset, and only declarations referenced immediately within a child will be written to a child's subset. This option is the default.
`rootallchildnone`	All entity declarations of the document tree should be written in the root document's subset only. Children will be saved without any entity declarations.
`roottreerefchildref`	The root document's subset will be written to include declarations for all entities referenced anywhere in the document tree. Childrens' subsets will

	be written to include only declarations referenced immediately within the child.
`roottreerefchildnone`	The root document's subset will be written to include declarations for all entities referenced anywhere in the document tree. Children will be saved without any entity declarations.
`rootrefchildref`	All files (objects) of the document tree will be written with only the declarations referenced immediately within that file (object).

If you are changing your site from using one of these modes to another, it is recommended that you load all documents and force the writing of all the entities of the document, before continuing with editing. To simplify this process, use the ACL alias `save_all_docs` on page 705. It will cause the root document and all recursively referenced file entities to be saved, whether or not the document/entity is marked as modified.

rootallchildall

The `rootallchildall` setting causes all entity declarations to be written in the root document's subset, and in the subset of all its children. This produces valid SGML in that the root document contains declarations for all entities of the document tree, and additionally writes all entity declarations in a special comment subset at the front of file entities. That can be useful if the file entities are edited standalone (that is, edited directly instead of being edited by being opened from the parent document) and if access to all the entities declared in the entire document tree is necessary during this standalone editing. With this setting, if an entity declaration is present in one file entity's subset, every file of that document tree that is saved will include the entity declaration.

The main drawback of this method is the larger size of fragments both when stored, and in memory. Another drawback is slower performance when performing operations such as deleting entity declarations.

While this setting most closely approximates pre-8.0 behavior, these drawbacks generally make it undesirable for sites where large documents are published of many file entities or objects.

Note

When a file entity is edited standalone, because it is the root document in that session, it will be saved according to the rules in effect for the root document.

rootallchildref

The `rootallchildref` setting, causes all entity declarations to be written in the root document's subset, and declarations for entities referenced immediately with a child to be written in the subset of each child. This produces valid SGML in that the root document contains declarations for all entities of the document tree, and additionally writes all entity declarations currently referenced by a file entity in a special comment subset at the front of the file entity. That can be useful if the file entities are edited standalone (that is, edited directly instead of being edited by being opened from the parent document).

This setting balances the need for standalone entity editing with the need for smaller entities. If it is not necessary to be able to access all entities while editing standalone, the smaller entity size, both when stored and in memory, makes this setting preferable to `rootallchildall`.

This option is the default.

Note

When a file entity is edited standalone, because it is the root document in that session, it will be saved according to the rules in effect for the root document.

rootallchildnone

The `rootallchildnone` setting, causes all entity declarations to be written in the root document's subset only, and no entity declarations to be written in file entity subsets. This produces valid SGML in that the root document contains declarations for all entities of the document tree. This is a good method when it is not necessary to edit entities standalone.

It may also be desirable if it is acceptable that all entity references are unrecognized when editing file entities standalone. The entity references will be preserved, but they will be treated as character entities, which may or may not invalidate the document structure, causing context errors. If you know that in your application it will not cause errors, then it might be a useful setting for you.

roottreerefchildref

The `roottreerefchildref` setting, causes entity declarations referenced anywhere in the document tree (that is, in the root document or its children) to be written in the root document's subset, and declarations for entities referenced immediately with a child to be written in the subset of each child. This produces valid SGML in that the root document contains declarations for all entities referenced in the document tree, and additionally writes all entity declarations currently referenced by a file entity in a special comment subset at the front of the file entity.

This option has all the benefits of `rootallchildref` and additionally purges the root document of declarations to unreferenced entities. It takes longer to save documents with this setting, so its primary value may be to periodically “clean” your documents.

Note

When a file entity is edited standalone, because it is the root document in that session, it will be saved according to the rules in effect for the root document.

roottreerefchildnone

The `roottreerefchildnone` setting, causes entity declarations referenced anywhere in the document tree (that is, in the root doc or its children) to be written in the root document's subset, and no entity declarations to be written in childrens' subsets. This produces valid SGML in that the root document contains declarations for all entities referenced in the document tree.

This option has all the benefits/considerations of `rootallchildnone` and additionally purges the root document of declarations to unreferenced entities. It takes longer to save documents with this setting, so its primary value may be to periodically “clean” your documents.

rootrefchildref

The `rootrefchildref` setting causes all files (objects) of the document tree to be written with only the declarations referenced immediately within that file. If an entity is referenced only from a file entity, not from the root document, this option produces invalid SGML in that the root document will not have declarations for all the entities referenced in the document tree. Such documents will produce SGML parsing errors if read with the check completeness (`-cc`) option.

set writenobreakattag

`set` writenobreakattag= { on | off | default}

This option determines whether line breaks are inserted at element boundaries or before the element close character (>) when writing and saving XML documents.

This option has a global scope. The settings are:

- `on` — Breaks will not be inserted at element boundaries and before element close characters (>).
- `off` — (The default.) Breaks may be inserted at element boundaries and before element close characters (>).

set writenonasciichar

`set` writenonasciichar= { entref | numref | char}

The `set writenonasciichar` command controls how Arbortext Editor writes out non-ASCII characters. The options are:

- `entref` — Arbortext Editor writes out non-ASCII characters as character entity references. (If a numeric entity reference had been inserted as a character, Arbortext Publishing Engine will write the numeric entity reference as a character.)

  If Arbortext Editor cannot find matching character entity references, it writes out the non-ASCII characters as characters in the target encoding.

  If Arbortext Editor cannot find the characters in the target encoding, it writes out the non-ASCII characters as numeric character references. This is the default setting.
- `numref` — Arbortext Editor writes out non-ASCII characters as numeric character references.
- `char` — Arbortext Editor writes out non-ASCII characters as characters in the target encoding. If Arbortext Editor cannot find the characters in the target encoding, it writes out the non-ASCII characters as numeric character references.

The abbreviation of `writenonasciichar` is `writenonascii`.

Note

The `write nonasciichar` command overrides the settings of this command.

When used with the `set entityinputconvert` and `set entityoutputconvert` commands, the `set writenonasciichar` also controls how Arbortext Editor writes out character entities.

Character Example

Arbortext Editor writes out the copyright character entity as a character (©) if the commands are set as follows:

```
set entityinputconvert=on
set writenonasciichar=char
```
or
```
set entityoutputconvert=on
set writenonasciichar=char
```

Numeric Reference Example

Arbortext Editor writes out the copyright character entity as a numeric reference (`&#xa9`) if the commands are set as follows:

```
set entityinputconvert=on
set writenonasciichar=numref
```
or
```
set entityoutputconvert=on
set writenonasciichar=numref
```

Entity Reference Example

Arbortext Editor writes out the copyright character entity as an entity reference (`&copy;`) if the commands are set as follows:

```
set entityinputconvert=on
set writenonasciichar=entref
```
or
```
set entityoutputconvert=on
set writenonasciichar=entref
```
or
```
set entityinputconvert=off
set entityoutputconvert=off
```
If either `entityinputconvert` or `entityouputconvert` is set to `on` and `writenonasciichar` is set to `entref`, Arbortext Editor converts entity references to characters and then converts them back to entity references. As a result of this conversion, the name of the entity reference that Arbortext Editor writes out may be different than the name of the entity reference that was read in.

However, if `set entityinputconvert` and `set entityoutputconvert` are `off`, Arbortext Editor preserves character entities and writes them out as entity references (it doesn't matter what the value of `set writenonasciichar` is). In this situation, the name of the entity reference that Arbortext Editor writes out is the same as the one that was read in since no conversion occurred.

set writepi

`set` writepi= { all | default | touchup | structural | none}

This option determines how the `save` on page 704, `write` on page 728, and `save_as` on page 705 commands treat Arbortext Editor processing instructions. This option has a global scope. The settings are:

- `all` — Saves Arbortext Editor processing instructions.
- `default` — Saves Arbortext Editor processing instructions except for <?Pub _display?>. This is the default setting.
- `touchup` — Saves Arbortext Editor processing instructions except for <?Pub _display?> and processing instructions that the editor uses to remember the state of the document display (for example, **<?Pub Caret?>**).
- `structural` — Saves the same Arbortext Editor processing instructions as the `touchup` setting except for <?Pub_font?>, <?Pub_newcolumn?>, <?Pub_ newline?>, <?Pub_newpage?> and <?Pub_nolinebreak?>.
- `none` — Removes Arbortext Editor processing instructions.

If the `-pi` or `-nopi` option is specified for the `write`, `save`, or `save_as` commands, then that value is used regardless of the `set writepi` option setting.

set writeunixfiles

`set` writeunixfiles= { on | off}

By default, the line delimiters written by Arbortext Editor are dictated by the `set outputlinebreak` command. Setting the `writeunixfiles` option to `on` forces Arbortext Editor to write UNIX line endings (line feed). Setting the `writeunixfiles` option to `off` (the default) resets the `outputlinebreak` option to its default value of `mswin`.

set writeunspecifiedattrs

`set` writeunspecifiedattrs= { none | fixed | all}

The `set writeunspecifiedattrs` command controls the writing of unspecified attributes. An unspecified attribute is any element attribute that does not have a user-defined value. Attributes that have a default value set in the Document Type Definition (DTD) but have not been set using the Arbortext Editor user interface are also considered unspecified.

The default setting is `none`, which means that Arbortext Editor will not write unspecified attributes when saving a file. When set to `fixed`, Arbortext Editor writes only the unspecified fixed attributes when saving a file. When set to `all`, Arbortext Editor writes all unspecified attributes, except for CDATA attributes, when saving a file. Note that the `fixed` and `all` settings both significantly increase the size of the written file.

Arbortext recommends that you do not use this command unless you need to write an instance of a file for an application that does not read attributes from a DTD. If you save a document with unspecified attributes and then re-open the document with Arbortext Editor, Arbortext Editor will display the attributes as local modifications.

set xmlextension

`set` xmlextension= *ext*

This command specifies the extension to use for newly saved XML files. `xml` is the default. The **Save As** dialog box and `save_as` command add this extension if the specified file does not already have an extension and the option is non-null.

A period should not be included as part of the extension.

Examples

```
set xmlextension=xml
set xmlextension="xml"
```

set xmlversion

`set` xmlversion= { 1.0 | 1.1}

This command controls the XML version used when creating and saving an empty document using the ACL function `doc_open` or the AOM **Application** interface `openDocument()` method. The XML version number specified is written on the XML declaration at the start of the empty document when it is saved. If set to `1.1`, it enables support for XML 1.1 characters in names and data. The default is `1.0` to use XML 1.0 rules.

XML 1.1 allows control characters in the range `#x1` through `#x1F` and `#x7F` through `#x9F`, most of which are forbidden in XML 1.0. If `xmlversion=1.1`, these characters are accepted and written as numeric character references, regardless of the `set writenonasciichar` setting, to conform to the XML 1.1 specification.

7

Hooks

Hook Functions

Hooks are user-defined functions that are called at strategic places by Arbortext Editor so you can customize editing operations. Setting a hook has global effect unless you build in additional user functions to specify a local level.

Refer to Hook functions introduction on page 115 for a detailed introduction to ACL hooks.

adapterstatehook

ret = **hook**(*op*, *adapterid*, *arr*[])

You register this hook using add_hook on page 216 to receive information about state changes from a repository adapter. The *op* argument is an operation code that indicates one of the following notifications:

- 1 — Repository adapter session connected
- 2 — Repository adapter session disconnected
- 3 — Repository adapter to give up user interface control

  This operation usually follows an adapter results available (4) operation and indicates that input focus should be taken.
- 4 — Additional repository adapter results are available

The *adapterid* argument is the identifier for the repository adapter. The *arr*[] argument is the name of an array of returned state information. It is currently only used for an adapter results available (4) operation and contains a list of Logical IDs on page 1036 that are selected in the repository adapter **Browser**.

The hook returns a return code *ret* with one of the following values:

- 1 — The operation completed successfully
- 0 — The operation was ignored
- -1 — The operation returned an error

catalogpathhook

catalogpathhook

Function prototype:

hook (*pathlist*)

Synopsis

Use with:
add_hook(*hookname*, *func*[, *prepend*])

`remove_hook(`*`hookname`*`, `*`func`*`)`

where *hookname* is `catalogpathhook`.

This hook is called whenever the `catalogpath` option is changed.

Argument

pathlist is a string giving the new list of directories to search for catalogs.

changetrackingaccepthook

`ret = `**`hook`**` (`*`oid1`*`, `*`oid2`*`, `*`nonacceptable`*`)`

This hook is called before change tracking markup is removed by a call to `change_tracking_accept_change` on page 238. The first and second arguments are *oids* of the one or two start tags used to mark up a single change track record. The first *oid* is suitable for a call to `change_tracking_info` on page 240. The *nonacceptable* parameter is set to `1` if the change record cannot be accepted, otherwise it is `0`.

The return value of the hook may have the following values:

- `-2` — to abort the operation with an error
- `0` — to perform the operation
- `-1` — to abort the operation without warning

changetrackingafterhook

`hook``(`*`oid1`*`, `*`oid2`*`, `*`optimized`*`)`

This hook is called after change tracking markup is inserted. The first and second arguments are *oids* of the one or two start tags used to mark up a single change track record. The first *oid* is suitable for a call to `change_tracking_info` on page 240. The *optimized* parameter is set to indicate the type of optimization done on the markup. `0` means that no optimization was done. Arbortext Editor will automatically optimize (or merge) two changes made by the same user to a single area of text or markup.

Note

If an operation causes multiple change tracking records to be inserted, then the `changetrackingafterhook` will be called for each one in the opposite order that they were inserted — that is, the last one is called first.

changetrackingrejecthook

`ret = hook (oid1, oid2, nonrejectable)`

This hook is called before change tracking markup is removed by a call to `change_tracking_reject_change` on page 240. The first and second arguments are *oids* of the one or two start tags used to mark up a single change track record. The first *oid* is suitable for a call to `change_tracking_info` on page 240. The **nonrejectable** parameter is set to `1` if the change record cannot be rejected, otherwise it is `0`.

The return value of the hook may have the following values:

- `-2` — to abort the operation with an error
- `0` — to perform the operation
- `-1` — to abort the operation without warning

chdirhook

`chdirhook`

Function prototype:

`hook=(wd)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `chdirhook`.

This hook is called after the working directory is changed by the `cd` command or by the **Open** or **Save As** dialog boxes.

Argument

wd is the path name of the new working directory.

compositionframeworkhook

`compositionframeworkhook`

Function prototype:

`hook(doc, type, where, params[])`

This hook is called at several points during the publishing operation. Each time the publishing framework calls the hook, it passes a different *where* value based on the point in processing. The hook function should check the *where* value.

The hook will be called for both Arbortext Editor and Arbortext PE server publishing. For Arbortext PE server publishing, the hook needs to be installed on both the client and server if the publishing operation runs a pipeline on the Arbortext PE server.

The publishing framework will invoke the hook function before `compose_for_type()` ACL functions start executing, after each operation `compose_for_type()` runs, and before `compose_for_type()` returns to its caller. Your hook function can return `0` to allow `compose_for_type()` to continue executing or return a negative value to terminate the publishing operation. It can modify any or all values in the publishing parameter array, thereby controlling the future flow of execution in `compose_for_type()`.

For DITA publishing, at least two pipelines will run, and the hook will be called several times during each pipeline run. Some run only on the client, others can run on either the client or the server. The DITA `createPrds` pipeline (a pre-process pipeline) always runs on Arbortext Editor client. The regular or post-process pipeline may run on the client or on the Arbortext PE server. Install a `compositionframeworkhook` on both to avoid problems with pipeline handling. Creating a hook function that can be installed on both client and server assures the appropriate action is performed, whether the processing is taking place on the client or on the server.

Note

It's possible the behavior of this hook can change from release to release. You should check your publishing framework hook code with each release of Arbortext software.

For more information on publishing framework processing and the `compose_for_type()` functions, refer to *The Publishing Framework* chapter of the *Programmer's Guide to Arbortext Publishing Engine*.

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `compositionframeworkhook`.

Arguments

- *doc* is the document identifier of the document being published.
- *type* is the name of the pipeline or type of published output as represented by its `.ccf` base file name.
  - `createPrds`

- dlmresolver
- foDoc
- htmlfile
- htmlhelp
- pdf
- profile
- rtf
- rtfDoc
- styler
- web
- xsl
- xslfo

The complete list of pipeline `.ccf` files is located in *`Arbortext-path`*`\composer`.

- *where* is the point in processing where the hook is called. It is a literal string value that is passed by the publishing framework to the hook function each time the framework calls it. For each string, there is a corresponding ACL variable that specifies the point of operation.

  `Arbortext-path``\packages\tools\composer.acl` contains the set of `HK_`*`variable-name`* ACL variables with defined values you can use for a *where* string specification. You can determine where the hook function is called by finding the variable associated with one of the *where* values and searching for it in `compose.acl`.

  For example, the first call to the publishing framework hook function is when the publishing framework starts. The *where* value is `initial`, which corresponds to the variable `HK_CFTI_INITIAL`.

```
global HK_CFTI_DIALOG_CB = "dialog_cb_complete"
global HK_CFTI_DITA_MAP = "dita_map_init_complete"
global HK_CFTI_DITA_POST = "dita_postproc_complete"
global HK_CFTI_DITA_TOPIC = "dita_topic_init_complete"
global HK_CFTI_FINAL = "final"
global HK_CFTI_INITIAL = "initial"
global HK_CFTI_PARAMETER_CB = "parameter_cb_complete"
global HK_CFTI_PE_DONE = "pe_done"
global HK_CFTI_PE_INIT = "pe_init"
global HK_CFTI_PE_POSTPROC = "pe_postproc"
global HK_CFTI_PE_SENT = "pe_sent"
global HK_CFTI_PIPE_DONE = "pipeline_done"
global HK_CFTI_PIPE_INIT = "pipeline_init_complete"
global HK_CFTI_PREPROC_CB = "preprocessing_complete"
```

```
global HK_CFTI_PREPROC_HOOK = "preproc_hook_complete"
global HK_CFTI_PROFILE = "profile_init_complete"
```

The complete list of variables and *where* values is located in `compose.acl`. Search for `HK_`.

- *params[]* is the parameter array that holds the parameters and values to be used by the pipeline.

completenesseventloghook

`completenesseventloghook`

Function prototype:

`function celhook(`*`step`*`,` *`message`*`,` *`errorFlag`*`,` *`linkType`*`,` *`linkTarget`*`,` *`linkName`*`)`

Synopsis

If completeness checking is invoked via either the `check_completeness` command or the `doc_incomplete()` function with option *4* or *8* set, this hook will be called each time Arbortext Editor finds a completeness error, just before the error is written to the completeness check event log.

Arguments

- *step* is the processing phase that produced the error:
  - `1`: completeness checking
  - `2`: markup checking
  - `3`: entity reference checking
  - `4`: ID and ID reference checking
  - `5`: schema check
  - `6`: table markup check
  - `7`: table markup warnings
- *message* is the actual error message
- *errorFlag* is 1 for errors and 0 for warnings
- *linkType* is 1 if *linkTarget* is an OID, 2 if *linkTarget* is an ACL function
- *linkTarget* is either an OID or an ACL function, as determined by *linkType*:
  - If it's an OID, it indicates the location in the document at which the error was located.
  - If it's an ACL function, it's a string that can be executed to obtain more information.

> **Note**
>
> The function will probably raise a dialog box, so this might not be useful if the programmer does not require UI components.

- The hook function can return 0 or -1. If it returns -1, the message is written to the completeness checking event log. If the function returns 0, the message is discarded and not written anywhere.

dcfreloadhook

`dcfreloadhook`

Function prototype:

```
hook(docid)
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `dcfreloadhook`.

This hook is called when a document type's configuration file (`.dcf`) is reloaded. This occurs when a `.dcf` file is saved by Arbortext Editor, when a `.dcf` file is saved externally to Arbortext Editor and a document of the document type is reverted, or when a `.dcf` file is loaded by the `set dcffile` command. The hook is called for each document of the document type.

docid is the document identifier of the document whose associated `.dcf` file was reloaded.

doc_create_hook

`doc_create_hook`

This hook is invoked whenever a document is created, such as when opening a document. This hook can be used with Arbortext Publishing Engine applications where running code in advance of a publishing run is difficult.

The following code gives an example of using this hook.

```
package myapp;
function doc_create_hook(doc)
{
 response(doc_path(doc));
 $ no return value is expected
```

```
}
add_hook("doccreatehook", "myapp::doc_create_hook")
```

editbeforehook

`editbeforehook`

Function prototype:

`hook (name)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `editbeforehook`.

This hook is called before the document or file specified by *name* is opened. If this function returns `-1`, the switch to a new document is canceled.

For the revert case (`edit -current`), `editbeforehook` is passed the path name of the document being reverted. The hook detects this case by comparing *name* to the current document's path name, `doc_path(0)`.

`editbeforehook` is called twice if the `edit` command is used without a path name (that is, to bring up the **Open Document** dialog box). `editbeforehook` is called with *name* equal to the null string before the dialog box appears, and then (if the first call does not return `-1` to abort the operation) called a second time with the path name selected from the dialog box.

Argument

name is the name of the document or file being opened.

editfilehook

`editfilehook`

Function prototype:

`hook (code)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `editfilehook`.

This hook is called whenever a file is edited, just after the document type instance files (`instance.acl` and `instance.js`) or document command files (`docname.acl` and `docname.js`), if any, are read.

Argument

code is an integer specifying detail about type of edit occurring:

- `0` — The document is being displayed into a new or empty window.
- `1` — The document is being reverted.
- `2` — The document replaces another document in the current window (for example, from an `edit` command without the `-newwindow` option, or a `doc_show` call).
- `3` — The document is being reverted from its untagged state to its tagged state.

editshowhook

`editshowhook`

Synopsis

Use with:

```
add_hook('editshowhook', func[, prepend])
remove_hook('editshowhook', func)
```

This hook is called before showing a document which is already open. If this function returns `-1`, the operation is canceled.

If the hook closes the already open document and returns `0`, the document will be opened.

Argument

path is the path of the document or file being opened.

entitydeclconflicthook

`entitydeclconflicthook`

Function prototype:

`hook (docid, dobj, entityname, entitytype, entitysubtype, pubid, sysid, notn, text)`

Synopsis

Use with:

```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `entitydeclconflicthook`.

This hook will be called when there is an entity whose declaration in the internal subset and one in the external subset (usually called the DTD).

Arguments

- *docid* is the document identifier of the document in which the declaration was found.
- *dobj* is the document object identifier of the document object in which the declaration was found.
- *entityname* is the entity name with either `&` or `%` prefix.
- *entitytype* is the entity type (that is: `file`, `text`, `graphic`, `char`, `filep`, `msp`, `accent`).
- *entitysubtype* is the entity subtype, that is:
  - for text — `CDATA`, `PI`, `MS`, `MD`, or empty string (normal)
  - for file — `SUBDOC`, or empty string (normal)
  - for graphic — for graphic — `NDATA`, `SDATA`, `CDATA`
- *pubid* is the point in processing where the hook is called. It is a literal string value that is passed by the publishing framework to the hook function each time the framework calls it. For each string, there is a corresponding ACL variable that specifies the point of operation.

  The complete list of variables and *where* values is located in `compose.acl`. Search for `HK_`.
- *sysid* is the parameter array that holds the parameters and values to be used by the pipeline.
- *notn* is the document identifier of the document being published.
- *text* is the document identifier of the document being published.

Note

The `file`, `filep`, and `graphic` entity types are external. The `text`, `msp`, `char`, and `accent` entity types are internal.

The hook function should return a string to use as the entity name, either without any prefix at all, or with the same prefix as was passed in (`&` or `%`). There are two cases with regards to the string returned:

1. If a blank string is returned, or if the string returned is the name of an already existing entity, or if the string returned is not valid as an entity name, then the conflicting declaration is simply ignored.
2. If the string returned is a valid entity name not matching an already existing entity, then that name is used for the new entity declaration and all references to that entity within the document are changed to the new name.

Note

If hook returns the name of an entity that is already declared, it is treated the same as if a blank string is returned. To avoid this, the hook should verify that the name being returned has not already been used. The built-in function `entity_exists` can be used to do this.

entityflattenhook

`entityflattenhook`

Function prototype:

`hook (`*`oid`*`)`

Synopsis

Use with:

```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `entityflattenhook`.

This hook will be called when flattening entity references in a document. The hook will be called each time the process encounters a region inside a `_file_ent` or `_text_ent` processing instruction tag pair.

Argument

oid is the oid of a processing instruction "tag pair" that wraps the contents of the entity, at the place where the flattened entity will appear in the saved document. This tag pair is the same as would wrap the entity if it were displayed inline using **View ► File Entities**, or **View ► Text Entities** (that is, `oid_name(oid)` will return either `_file_ent` or `_text_ent`.

If the hook returns -1, that entity reference will not be flattened, but will be left displayed inline, wrapped by the `_file_ent` or `_text_ent` tag pair. Any other return value will result in the entity being flattened.

Note

It is not possible for the hook to alter the wrapper pair or the contents of the entity (as it is a protected region). The hook can, however, either delete the entire entity (in which case it must return `-1`), or insert custom processing instructions around the wrapper pair so after the wrapper pair is deleted, you will retain information about former entity boundaries.

entitylockhook

`entitylockhook`

Function prototype:

`hook (oid, entname)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `entitylockhook`.

This hook will be called before an entity is locked, either explicitly (for example, by using the `oid_entity_lock` on page 438 function) or implicitly (for example, by typing in the expanded entity). It will only be called if the entity is a locally stored file (entity files stored in a Document Management System cannot be locked this way). It can override or cancel the locking operation similar to using the `editbeforehook` to detect when the user is about to edit an entity in a separate window.

If the lock operation is not a result of inline editing, then the *oid* parameter will be null. If this hook returns `0` the code proceeds as if the hook was not called and locks the entity. If it returns `-1` the lock operation is cancelled and the attempted locking operation fails, unless the hook locked the entity itself.

Arguments

- *oid* is the oid of the reference to the entity.
- *entname* is the name of the entity being locked.

formatbeforehook

`formatbeforehook`

Function prototype:

`hook (COMMAND, want_final)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `formatbeforehook`.

This hook is called immediately before the first formatting pass when printing, previewing, or formatting a document. The *want_final* flag indicates that the command that caused formatting (`print`, `preview`, `format`, and so on) was requested to do as many formatting passes as necessary to produce the a final version of the document. For example, if the command that started formatting was `preview allpasses`

`formatbeforehook` will be invoked with the *command* parameter set to `preview` and the *want_final* parameter set to `1` before each formatting pass.

This hook is allowed to modify the document. When modifying the document, generated text is recalculated and the `.tex` file is rewritten. This hook cannot be used to invoke formatting.

Returning `true` writes a new `.tex` file. Returning `false` allows a `.tex` file write but does not force the write. Returning `true` is appropriate if the hook changes state in such a way that it would affect the result of system functions called from the stylesheet. Typically, a `.tex` file is rewritten only if the document instance changes or if generated text must be recalculated. The hook assumes formatting is being done on `current_doc()`.

Arguments

- *command* is a string specifying the command requesting formatting: `format`, `preview`, or `print`.
- *want_final* is `1` if all formatting passes are specified. Otherwise, *want_final* is `0`.

formatcompletehook

`formatcompletehook`

Function prototype:

`hook (pageno)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `formatcompletehook`.

This hook is called when document formatting completes after print or preview processing, if any, has been initiated.

Note

This hook must not launch a modal dialog box (for instance, by using the `response` function). Problems may result if the *-wait* modifier is used with a formatting command.

Argument

pageno is the ordinal page number of the last page formatted.

formatcontinuehook

`formatcontinuehook`

Function prototype:

`hook (pageno, doc)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `formatcontinuehook`.

This hook is called after formatting has completed, but before print processing or previewing is initiated. It allows you to modify the document and request additional formatting passes.

Note

This hook function should not use the `preview`, `format`, or `print` commands.

Arguments

- *pageno* is the ordinal page number of the last page formatted.
- *doc* is the document being formatted.

Return Values

This hook function should return one of the following values:

- `0` — no more formatting is necessary; continue with normal post-format processing (e.g. initiate printing, update preview window)
- `1` — perform one additional formatting pass (after which this hook will be called again)
- `2` — perform as many additional formatting passes as necessary for final output (after which this hook will be called again)
- `3` — perform additional formatting passes using the `-onepass|allpasses` modifier on the original formatting command on page 653 to determine how many to do (after which this hook will be called again)

The output of the additional passes will reflect any changes made to the document by the hook function.

formaterrorhook

`formaterrorhook`

Function prototype:

`hook (`*`errorFlag`*`, `*`pageNumber`*`, `*`passNumber`*`, `*`message`*`)`

Synopsis

Use with:

```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `formaterrorhook`.

This hook is called if formatting produces an error. If the ACL `set formatwarnings` option is set to `off`, this function will not be called for format warning messages.

Arguments

- *errorFlag* is `1` for an error and `0` for a warning.
- *pageNumber* is the page being generated when the error or warning is detected, or is `-1` if no page number can be determined.
- *passNumber* is the formatting pass number. The first pass is number zero.
- *message* is the actual error message.

formatpagestatushook

`formatpagestatushook`

Function prototype:

`hook (doc, pageno )`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `formatpagestatushook`.

This hook is called when document formatting starts formatting a new page. Just before formatting begins, this hook is called with a *pageno* of `zero (0)`.

Arguments

- *doc* is the document identifier of document being formatted.
- *pageno* is the ordinal page number of the page about to be formatted.

graphicpathhook

`graphicpathhook`

Function prototype:

`newname = hook (filename, oid)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `graphicpathhook`.

This hook is used to resolve a graphic file name to display the graphic image in the **Edit** pane.

Arguments

- *filename* is the path and file name of the graphic as determined from the document instance, after any entity path name resolution has been done. The hook may substitute a different file name by returning its path and file name. Otherwise, the hook must return *filename*
- *oid* is the object ID of the graphic tag identified by *filename*.

htmldoccompletehook

`htmldoccompletehook`

Function prototype:

`hook=(htmldocid, srcdocid)`

Synopsis

Use with:
```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `htmldoccompletehook`.

Arbortext Editor calls this hook before returning a virtual HTML document generated from `htmldoc` on page 380 function. The hook passes the document's virtual HTML document ID and the associated source document ID to the function *func*. You can perform any type of supported Arbortext Editor action on the document while it's in memory, such as finding and replacing text. The function is especially useful for changing markup in HTML elements or adding new tags.

Arguments

- *htmldocid* is the document identifier of the virtual HTML document returned by `htmldoc`.
- *srcdocid* is the document identifier of the source document.

The hook function should perform actions that conform to valid standardized HTML markup.

htmlfloathook

`htmlfloathook`

Function prototype:

`hook=(floatnum, fclassname, floattype, src_oid, a_oid, div_oid))`

Synopsis

Use with:
```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `htmlfloathook`.

Arbortext Editor calls this hook when it encounters a float while converting a document to HTML. The hook passes the float's arguments (described below) to the associated function *func*.

Arguments

- *floatnum* is an integer unique to each float, in order of the float's anchor, such that the first float encountered in the in the document is 1, the next 2, and so on.
- *fclassname* is a string which is the floatloc floatid from the FOSI, or in the case of footnotes, it is the string `__footnote`.
- *floattype* is an integer representing the floatloc floattyp from the FOSI where `0` means once, `1` means atpgbrk, `2` means atcolbrk, and `3` means multiref.
- *src_oid* is the oid in the SGML or XML document which caused the float per the FOSI. This makes it possible to customize behavior based on which element floated (e.g., a graphic or a table) or based on one of its attribute values, its context, etc.
- *a_oid* is the oid of the a element which is inserted into the output virtual document (vdoc). It is the responsibility of the ACL code to assign an href to this element which points to the float, and to insert whatever content is to be used for the hot spot.
- *div_oid* is the oid of the `div` element which is placed around all of the content for a given float in the output vdoc. Unless the float is to be left inline, the function called by the hook should select and cut this oid and its content, and paste or write it elsewhere.

htmlimginserthook

`htmlimginserthook`

Function prototype:

`hook=(`*`oid,filename`*`)`

Synopsis

Use with:

```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `htmlimginserthook`.

Arbortext Editor calls this hook when it encounters a graphic or equation while converting a document to HTML using the `htmldoc` on page 380 function. The hook passes the graphic or equation's OID and file name to the associated function *func*.

Note

Use this hook to convert graphics to file formats that are supported by common Web browsers, such as GIF, JPEG, and PNG.

Arguments

- *oid* is the object identifier (OID) of the graphic or equation.
- *filename* is the complete path and file name of the graphic or equation.

  The hook function should:

  - Verify that the graphic or equation has an appropriate file format extension.
  - Verify that the file name being returned has not been used in the document.
  - Prefix a relative or absolute path to the graphic or equation's file name.

The hook function should return a string that Arbortext Editor will assign to the *SRC* attribute for the respective `IMG` element in the HTML document.

includeflattenhook

`includeflattenhook`

Function prototype:

`hook (`*`dobj`*`)`

Synopsis

Use with:

```
add_hook(hookname, func)
remove_hook(hookname, func)
```

where *hookname* is `includeflattenhook`.

This hook will be called when flattening XML inclusions in a document. The hook will be called each time the process encounters an XML inclusion.

Argument

dobj is the document object representing the inclusion to be flattened.

If the hook returns −1, that inclusion will not be flattened, but will be left displayed inline. Any other return value will result in the inclusion being flattened.

includelockhook

`includelockhook`

Function prototype:

`hook (oid, entname)`

Synopsis

Use with:
`add_hook(hookname, func[, prepend])`
`remove_hook(hookname, func)`

where *hookname* is `includelockhook`.

This hook will be called before an XML inclusion is locked, either explicitly (for example, by using the `oid_entity_lock` on page 438 function) or implicitly (for example, by typing in the expanded inclusion). It will only be called if the inclusion is a locally stored file (XML inclusion files stored in a Document Management System cannot be locked this way). It can override or cancel the locking operation (similar to using the `editbeforehook` on page 946 to detect when the user is about to edit an entity or inclusion in a separate window).

If this hook returns `0` the code proceeds as if the hook was not called and locks the entity.

If it returns `-1` the lock operation is cancelled and the attempted locking operation fails, unless the hook locked the entity itself.

Arguments

- *oid* is the oid of the reference to the XML inclusion.
- *entname* is the name of the inclusion being locked.

ixkeycharenthook

`ixkeycharenthook`

Function prototype:

`hook (oid)`

Synopsis

Use with:
`add_hook(hookname, func[, prepend])`
`remove_hook(hookname, func)`

where *hookname* is `ixkeycharenthook`.

The index key character entity hook handles a subset of the problem addressed by the index head to key hook. It allows the user to customize the strings used to replace symbols in index headings.

Argument

oid is the oid of the desired character entity reference entity name.

The hook function must return a character string to use in the index key in place of the character entity. If an empty string is returned, it means to ignore this entity for sorting purposes. To have default character entity mapping done for an entity, the hook should return the results of the `ixkey_charent` on page 390 function.

This is useful when the character entity mapping that Arbortext Editor does based on the configuration files in the *`Arbortext-path`*`/lib/ixlang` directory of the installation tree is not sufficient. For example, your mapping may need to be conditional on the context of an entity's use.

ixkeymarkuphook

`ixkeymarkuphook`

Function prototype:

`hook (`*`oid`*`)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `ixkeymarkuphook`.

Normally, elements and pseudo-tags inside index headings do not affect sorting. (However, if they have generated text in the context of the preliminary index, the generated text does affect sorting.

The index key markup hook allows the user to have markup affect the conversion of an index heading to an index key. Again, this is a subset of the index heading to key conversion process that can be handled by the index head to key hook.

Argument

oid is the oid of a start tag.

The hook function must return a character string to use in the index key in place of the markup. If an empty string is returned, it means to ignore this markup for sorting purposes.

keybaselistchangedhook

`keybaselistchangedhook`

Function prototype:

`hook (doc, path)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `keybaselistchangedhook`.

This hook will be called any time a DITA document's `ditakeybaselist` option is updated.

Arguments

- *doc* is the ACL ID of the document for which the `ditakeybaselist` option is updated.
- *path* is the updated list of paths.

keyrefResolveHook

`keyrefResolveHook`

Function prototype:

`hook (key, type, oid, flag)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `keyrefResolveHook`.

This hook will be called any time the *keyref* or *conkeyref* attribute on an element in a DITA document is being resolved to an absolute reference.

Note

A legitimate value of an empty string must be distinguished from an error returned if the key cannot be resolved. If an empty string is to be returned, the special value of `*ATI_EMPTY_STRING*` should be returned. Returning an empty string `""` indicates that the attribute cannot be resolved.

Arguments

- *key* is the key name that is being resolved.
- `type` is the type of reference that is being resolved. Valid values are `keyref` and `conkeyref`
- *oid* is the object identifier (OID) of the element for which the attribute value is being resolved.
- *flag* provides additional information about the type of resolution. If this is set to `0x001`, that indicates the attribute is being used for publishing. If this is not set, that indicates the attribute is being used for editing.

menuloadbeforehook

`menuloadbeforehook`

Function prototype:

`hook(`*`window`*`, `*`cmd`*`)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `menuloadbeforehook`.

This hook is called before a new set of menus is loaded into the menu bar by a `menu_load` command or when a new document is loaded.

Note

This contrasts with the `menuloadhook` on page 962 function which is called after the menu bar has been loaded.

The hook function can be used to substitute a different set of menus by issuing a recursive `menu_load` command and returning `-1` to signal that the hook function handled the menu load.

Arguments

- *window* is the window identifier of the window containing the menu bar.
- *cmd* is the full `menu_load` command string that will be executed. The hook can modify this command (or replace it) and execute it with the `execute` on page 351 function. It should return `-1` in this case.

menuloadhook

`menuloadhook`

Function prototype:

`hook(`*`window`*`, `*`filename`*`)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `menuloadhook`.

This hook is called after a new set of menus is loaded into the menu bar by a `menu_load` command or by Arbortext Editor when loading a new document. Note, this contrasts with the `menuloadbeforehook` on page 961 function which is called before the menu bar has been loaded. Arbortext Editor will not reload menus when switching documents if no menu changes were done. The hook function can be used to add a custom menu to the standard menus using `menu_add` commands without the need to change the system menu configuration file, `editmenu.cf`.

Note

You should always place commands that modify menus in edit class windows in a `menuloadhook` function. Doing so ensures that the `menu_add`, `menu_change` and `menu_delete` commands are executed every time the menus are reloaded, not just when a document is first read. Some user actions force the reloading of menus, such as switching between **Full Menus** and **Short Menus**.

Modifying default menus

You can do the following to modify default menus:

- For a particular document type, place the commands to modify or load menus in the document type instance command files on page 48 (`instance.acl` and `instance.js`).
- For a specific document, place the commands to modify or load menus in the document command files on page 48 (*`docname`*`.acl` and *`docname`*`.js`) in the directory containing the document *`docname`*.

Arguments

- *window* is the window identifier of the window containing the menu bar.
- *filename* is the path name of the menu configuration defining the menus. It is null if the system configuration is not read and the default set of menus were loaded.

newfilehook

`newfilehook`

Function prototype:

```
hook ()
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `newfilehook`.

This hook is called before the **New Document** panel is displayed to open a new document.

parsererrorhook

`parsererrorhook`

Function prototype:

```
hook(err, msg, file, line)
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `parsererrorhook`.

This hook is called before a parser error or warning is output. The hook function is passed *err*, a boolean specifying `1` if *msg* is an error and `0` if *msg* is a warning, and *msg* is a string containing the parser message. *file* is the name of the file to which the message applies, and *line* is the line number in the file producing the error.

If the hook function returns `-1`, the message is suppressed. Otherwise, the message is output as usual.

In this example, the hook function suppresses warnings but not errors.

```
function ParserErrorHook(err, msg)
{
   if (! err)
      {
      return -1; # suppress message
      }
   return 0;
}
add_hook("parsererrorhook", "ParserErrorHook");
```

postexporthook

`postexporthook`

Function prototype:

`hook(outFile)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `postexporthook`.

The post export hook is called after a successful export operation. The input parameter is the path name of the exported document. The hook is not called if the export fails. The return value of the function is ignored.

Argument

outFile is the path name of the exported document.

postimporthook

`postimporthook`

Function prototype:

```
hook(doc)
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `postimporthook`.

The post import hook is called after the document has been successfully imported, but before the document is displayed in a window (if not batch mode). The input parameter is the document identifier of the imported document. The hook is not called if the import operation fails. If the hook returns `-1`, the document will not be displayed in a window (if not batch mode). It is the application's responsibility to close the document in this case.

Argument

doc is the document identifier of the resulting document.

preexporthook

`preexporthook`

Function prototype:

```
hook(arr[])
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `preexporthook`.

This hook is called before the export process begins, just after the user supplies the desired parameters on the **Export Document** dialog box. The hook function is passed an array containing all the input parameters as specified on the `document_export` function (or from the dialog box), and may override the settings by altering the array. If the hook function returns `-1`, the export operation will be cancelled, causing `document_export` to return `0`. If the hook modifies any of the parameters in the array, it must return `1`. Otherwise, the function should return `0`.

Argument

arr is an array parameter whose elements contain the `document_export` parameters indexed as follows:

- `1` — the input path name *inFile*
- `2` — the output path name *outFile*
- `3` — the output file format *outFileFmt*
- `4` — the map file path name *mapFile*
- `5` — the style template file path name *styleTmplt*
- `6` — the log file path name *logFile*
- `7` — the boolean *batchMode* value

Example

In the following example, the hook function forces the export process to use a different style template by changing the *styleTmplt* parameter.

```
function PreExportHook(arr[])
{
   arr[5] = "//server/templates/company.dot";
   return 1; # signal array was changed
}
add_hook("preexporthook", "PreExportHook");
```

preferencehook

`preferencehook`

Function prototype:

`hook=(win)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `preferencehook`.

This hook is called just before the **Preferences** dialog box is displayed. Use this hook to make changes to the dialog box before it displays.

Argument

win is the window identifier for the desired window.

Example

In the following example, the `preferencehook` callback invokes the `prefhook` function, which hides the **Full Menus** item on the dialog box.

```
function prefhook(win)
{
```

```
   dlgitem_hide(win, "fullmenus");
   return 0
}
add_hook("preferencehook","prefhook")
```

previewlinkhook

`previewlinkhook`

Function prototype:

`hook (`*`pageno`*`)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `previewlinkhook`.

The `previewlinkhook` function is called after Arbortext Editor repositions the cursor in the Edit pane due to a request from Print Preview to match the text at the cursor in the Edit pane to the text under the pointer in the **Print Preview** window. You can get the repositioned location within the document using the `oid_caret` on page 430 function.

Argument

pageno is the ordinal page number displayed in the **Print Preview** window.

printcompletehook

`printcompletehook`

Function prototype:

hook ()

Synopsis

This hook is called when a **Print Composed** request is finished.

profiledochook

`profiledochook`

Function prototype:

`hook(`*`doc`*`)`

Synopsis

Use with:
`add_hook(`*`hookname`*`, `*`func`*`[, `*`prepend`*`])`
`remove_hook(`*`hookname`*`, `*`func`*`)`

where *hookname* is `profiledochook`.

The `profile_config()` on page 486 built-in function calls this hook to allow an alternate profiling configuration document to be specified in place of the one specified in the `.dcf` file.

Argument

doc is the document identifier of the instance being loaded.

The hook function can return `-1` to disable profiling for the document, `0` to use the default profile document as returned by `profile_config()`, or the document identifier of a profiling configuration document loaded by the hook function. The hook function may call `profile_config()` if necessary.

tblmodelprompthook

`tblmodelprompthook`

Function prototype:

ret = `hook (`*`tmIDs[]`*`)`

Synopsis

Use with:
`add_hook(`*`hookname`*`, `*`func`*`[, `*`prepend`*`])`
`remove_hook(`*`hookname`*`, `*`func`*`)`

where *hookname* is `tblmodelprompthook`.

If the current document type supports multiple table models, the `tblmodelprompthook` function is called when a user inserts a table and the cursor is in a location in which it is valid to insert more than one table model. This hook lets users choose the table model they want to insert.

Arguments

tmIDs is an array of valid table model IDs.

Return Arguments

- `-1` — Cancels the table insertion.
- `0` — Displays the prompt if `set prompttablemodels` on page 878 is `on`, or automatically inserts the primary table model if `set prompttablemodel` is `off`.
- Non-zero integer — A one-based index value into the *tmIDs* array that indicates the table model to be inserted.

untrackedchangehook

`ret = hook (doc, type)`

This hook is called before a change is made to the document that will not be tracked. In general, the change is permitted but a warning is issued. *doc* is the document in which the change is being attempted. *type* is a string containing the name of the attempted operation (command or function) that is being called.

The return value of the hook `ret` may have the following values:

- `0` — to perform the operation and warn the user
- `-1` — to silently abort the operation
- `-2` — to abort the operation with an error

userulehook

`userulehook`

Function prototype:

`hook (window useruleid useparam oid value state docid)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `userulehook`.

The `userule` hook allows the user to customize processing of FOSI usetexts that have non-zero userule values. It can be used either to customize Arbortext Editor's built-in userule processing (indexing and userule exporting), or to perform entirely different custom processing. The `userule` hook can be used for any of the following purposes:

- To alter the preliminary index document and then invoke index processing
- To change an ACL variable or set option that affects index processing or export handling

- To alter the contents of a usetext before it is inserted as generated text
- To conditionally decide whether a usetext should be inserted as generated text
- To replicate built-in processing for exporting and indexing

The hook is called:

- when generated text is updated
- before formatting for published output
- after a formatting pass if either of these happens:
  - page number references within the usetext were changed by the formatting pass
  - previous calls to the hook returned `-1` (see the explanation of the *state* argument)

Arguments

- *window* is the integer id of the window in which index processing occurs. The hook is not expected to use this value except as a "handle" to pass to the `indexproc` and `fosivar_value` ACL functions.
- *useruleid* is the integer userule id (sequence number). This is also a "handle" to be passed to the `indexproc` ACL function.
- *useparam* is the string value for the useparam.
- *oid* is the oid of the element being processed in the user's document instance whose element-in-context (e-i-c) contains the usetext with the non-zero userule.
- *value* is the integer userule value.
- *state* is the integer indicating userule state. If the state is `1`, this is the first time that the `userulehook` is being called for this *oid* and this specific usetext (since it is possible for an *oid* to have more than one usetext). The contents of the document referenced by *docid* are from the first pass of formatting. If the state is `2`, the `userulehook` is being called again for this *oid* and this specific usetext and the content of the document referenced by *docid* is different from the previous call. Usually this is because some page numbers have changed. If you are trying to save the contents to a file or drive some external system, you may want to reprocess because of the changes. If the state is `3`, the `userulehook` is being called again for this *oid* and this specific usetext and the content of the document referenced by *docid* has not changed. If you are trying to save the contents to a file or drive some external system, you do not have to reprocess as the contents have not changed.

The hook will only be called for state `3` if the hook returns `-1` when called for state `1` and `2`. This mechanism is useful when exporting files as it will cause the file only to be written when page numbers have stopped changing.

- *docid* is the integer id of the document containing the preliminary content that can be modified.

The hook function must return `1` or `-1`. `-1` means do not insert the usetext contents into the output stream; `1` means to insert.

If the preliminary document is modified, and it is desired to then insert the preliminary document into the published output stream, the function must return `1`. If nothing is to be inserted in the published output stream, the function must return `-1`. (This can be useful if the preliminary userule is written to a file, which is referenced elsewhere in the document or is otherwise processed.)

When using the name of an index coding pseudo-element (perhaps when searching for tag occurrence in the document), you must append an asterisk to the pseudo-element name, (for example, `ixpt*`).

Note

If the userulehook causes a cross reference to be inserted into the document, the hook must include the following line of ACL for the cross reference to be properly resolved:

main::RESOLVE_XREFS_AFTER_USERULES=1

Examples

The following example shows how to alter the preliminary index document and then invoke index processing using the `indexproc` built in function. This sample code returns the result of `indexproc`.

```
# example userulehook: change index to uppercase
function simpleuserulefunc(view_id, userule_id, \
userule_param, userule_oid, userule_value, userule_state, \
userule_doc_id)
{
   local otop, obot
   local savedoc = current_doc(userule_doc_id)
   # translate the prelim doc to upper case
   otop = oid_first(userule_doc_id)
   obot = oid_last(userule_doc_id)
   goto_oid(otop)
   mark begin
   goto_oid(obot)
   mark end
   translate uc
   # convert the prelim index doc into the final index doc
```

```
   indexproc(view_id, userule_id, userule_param)
   current_doc(savedoc)
   return 1
}
```

This example shows how to replicate built-in processing for exporting and indexing, and how to add custom processing for userule value 3:

```
function process_userule3(predoc_postdoc)
{
   ...
}
function ur(view_id, userule_id, userule_param, \
userule_oid, userule_value, userule_state, \
userule_doc_id)
{
   local otop, obot
   local savedoc = current_doc(userule_doc_id)
      if (userule_value == 1 && userule_state < 3)
         {
         Local filename
         filename = "index1.sgml"
         write -ok -sgml -nopi -noheader $filename
         current_doc(savedoc)
         return -1
         }
      else if (userule_value == 2)
         {
         indexproc(view_id, userule_id, userule_param)
         current_doc(savedoc)
         return 1;
         }
      else if (userule_value == 3)
         {
         process_userule3(userule_doc_id)
         current_doc(savedoc)
         return 1
         }
   return -1;
}
```

writetexafterhook

`writetexafterhook`

Function prototype:

`hook (`*`command`*`, `*`final`*`)`

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `writetexafterhook`.

This hook is called before the `.tex` file is written to restart the formatter.

Arguments

- *command* specifies the name of the command requesting formatting: `format`, `preview`, or `print`.
- *final* is `1` if final formatting was specified, otherwise is `0`.

writetexhook

`writetexhook`

Function prototype:

```
hook (command, final)
```

Synopsis

Use with:

```
add_hook(hookname, func[, prepend])
remove_hook(hookname, func)
```

where *hookname* is `writetexhook`.

This hook is called before the `.tex` file is written to restart the formatter.

Arguments

- *command* specifies the name of the command requesting formatting: `format`, `preview`, or `print`.
- *final* is `1` if final formatting was specified, otherwise is `0`.

8

Callbacks

Callback Functions

Callback functions are user-defined functions called during specific events in Arbortext Editor that allow you to customize editing operations. Callback functions provide both a global and a local level of registration or notification to isolate events.

Refer to Callback Functions Introduction on page 112 for a detailed introduction to ACL callbacks.

channel_set_callback

channel_set_callback (*ch*, *funcname*[, *data*])

This function sets the specified function *funcname* as a callback to be invoked whenever certain events are detected on the network channel *ch*. *ch* is a channel identifier returned by `open_connect` or `open_accept`. The optional argument *data* is evaluated and its value is passed to the specified callback function.

Calling `channel_set_callback` automatically sets the channel *ch* to non-blocking mode.

The callback function must be declared as:
`function funcname(ch, op[, data])`

where *ch* is the channel identifier, *data* is the user data argument passed to `channel_set_callback`, and *op* is the integer specifying the type of notification.

- `0` — the channel was opened. This notification is actually given when data are first available for reading and allows the callback to initialize its state.
- `1` — the channel was closed. Note, that if `read` returns 0 when called on a read (op=2) notification, Arbortext Editor gives a close notification instead of returning 0 from `read`.
- `2` — the channel is ready for reading (that is, there is more incoming data).

The callback must call `read` (or `getline` if appropriate) on a read notification to read data from channel. The callback cannot be invoked again until after a read is performed. The callback can call `read` more than once on a single notification, but must be prepared to deal with a partial or incomplete read.

The callback is automatically removed when the channel is closed.

The sample function that follows uses `open_connect` and `channel_set_callback` to fetch an HTML document from an HTTP server (well known port 80). For clarity, it does minimal error checking and does not deal with the MIME headers that are normally transmitted preceding the actual document. Unlike the

similar example given with `open_connect`, this uses `channel_set_callback` to read from channel in a non-blocking manner (that is, the user function `ch_notify` gets called only when data are available on the channel).

```
function http_async_get(host, path, lclfile) {
   local of = open(lclfile, "wb")
   if (of < 0) {
      message "http_get: couldn't open $lclfile for write"
      return 0;
   }
   local ch = open_connect(host, 80)
   if (ch < 0) {
      message "http_get: $main::api_error"
      return 0;
   }
   channel_set_callback(ch, "ch_notify", of)
      write(ch, "GET " . path . " HTTP/1.0\r\n\r\n")
      return ch
   }
   function ch_notify(ch, what, of)
   {
      switch (what) {
      case 0: # open
         # could open the output file here
         break;
      case 1: #close
         close(ch)
         close(of)
         break;
      case 2: # read
         local buf, len
         len = read(ch, buf, 512)
        switch (len) {
        case -2: # read would block
         return;
      case -1: # unexpected error
         message "ch_notify: $main::api_error"
         # fall through
      case 0: # connection was closed
         # note, by closing CH here this callback
         # gets removed so that we won't get a
         # subsequent close notification (1).
         close(of)
         close(ch)
         break
      default:
         write(of, buf, len)
        }
     }
   }
```

dlgitem_add_callback

dlgitem_add_callback (*window dlgitem*, *callback*[, *"PREPEND"*])

Appends *callback* to the list of callback functions to be called when the value of *dlgitem* within *window* changes. Functions are called in the order that they appear in the list. If successful, the function returns `1` (one).

`dlgitem_add_callback` has the following parameters:

- *window* — The window identifier. If *window* is invalid, *$ERROR* is set and `0` is returned.
- *dlgitem* — The value of the control's *id* attribute. If *dlgitem* does not exist within window, *$ERROR* is set and `0` is returned.
- *callback* — Specifies the actual callback to add. If *callback* is already on the callback list for the dialog item, the list is reordered to place the callback in the new position. The callback is not added to the list a second time.
- *"PREPEND"* — Optional. When specified, the callback is added to the beginning of the list of callback functions.

Three different events are associated with dialog items:

- `ITEM_CHANGED`
- `ITEM_FOCUSED`
- `ITEM_UNFOCUSED`

The callback is called if any of these events occurs. For example, clicking on a button may generate two callbacks, once when the button takes focus, and again when clicked (that is, changed). Therefore, it is a good practice for your callback function to execute only if the desired event has occurred. Otherwise, it should simply return.

Example:

```
$ret = dlgitem_add_callback($win, "TextField",
       main::TextFieldCallback);
```

Callback prototype:

```
callback(windowId, dialogItem, eventType, eventSubType, detail)
```

Arguments:

- *windowId* — The identifier of the window that contains this dialog item.
- *dialogItem* — The value of the dialog item's ID attribute.
- *eventType* — One of the following three types.
  - `"ITEM_CHANGED"`
  - `"ITEM_FOCUSED"`

- "ITEM_UNFOCUSED"

- *eventSubType* — Applies only when *eventType* is "ITEM_CHANGED". The following controls have the following *eventSubType* values:
  - combobox control
    - "SCROLL" — Dispatched when the text field in the combobox is changed.
    - "SELECT"
    - "ACCEPT" — Dispatched when the **ENTER** key is pressed. The event is sent out for simple and dropdown comboboxes. It is not sent out for the dropdownlist comboboxes.
    - "OPEN" — (Dropdown combobox only.)
    - "CLOSE" — (Dropdown combobox only.)
  - listbox control
    - "DOUBLE_CLICK"
  - toolbar list dropdown control
    - "POPULATE" — Dispatched when the list dropdown is open.
    - "SELECT"
  - tree control
    - "DOUBLE_CLICK"
- *detail* — Applies only when *eventType* is "ITEM_CHANGED" and *eventSubType* is "CLOSED". If the combobox dropdown is closed using the **ESC** key, the value of *detail* is CANCEL. If the combobox dropdown is closed by other means, *detail* is an empty string.

dlgitem_remove_callback

dlgitem_remove_callback(*window*, *dlgitem*, *callback*)

Removes *callback* from the list of functions called when the value of *dlgitem* within *window* changes value. If *window* is invalid or *dlgitem* does not exist within *window*, or *callback* is not on the callback list for the dialog item, $ERROR is set and 0 is returned. If successful, the function returns a one (1).

The *window* parameter is a window identifier. *dlgitem* is the value of the control's *id* attribute. The *callback* parameter specifies the actual callback to remove.

Example

```
$ret = dlgitem_remove_callback($win, "TextField",
       main::TextFieldCallback)
```

doc_add_callback

doc_add_callback (*doc*, *cbtype*, *callback*[, *"PREPEND"*])

Appends callbacks to the list of callback functions to be called when a particular document-level event occurs. Two lists of callbacks are maintained for each callback type, one is document specific and the other is global (applies to all documents).

The *doc* parameter is a document identifier or zero (0). Using a value of zero adds the callback globally (for all documents). The *cbtype* parameter is the name of a callback type. The callback parameter specifies the callback function to add. The *"PREPEND"* flag causes the callback to placed at the beginning rather than the end of the list. Callback functions are called in the order that they appear on the list, global callbacks are called first, followed by document specific callbacks.

If *doc* is invalid, *$ERROR* is set and 0 is returned. If successful, the function returns a unique positive document identifier.

Note

If *callback* is already on the callback list for the document, the list is shuffled to place the callback into the desired position; the callback is not added to the list a second time.

Note

Arbortext Editor supports a single callback of each type per document. For example:

```
$retval = doc_add_callback(current_doc(), 'insert_entity', 'ietest')
function ietest (doc, entity, op) {
 response("doc = " . doc . "; entity = " . entity . "; op = " . op)
}
```

Refer to Callback Functions introduction on page 112 for helpful overview information. Also, refer to the following valid document callback types:

attribute_default_value on page 981
clone on page 983
completeness_check on page 983
conref_content on page 983
context_changed on page 984
modify_tag on page 1003
paste on page 1004
pending_delete on page 1004
pending_delete_after on page 1005
print_panel on page 1005

attribute_default_value Callback Type

Function prototype:

```
function funcname (doc, oid, attrArr[,flag])
```

`attribute_default_value` is called whenever default attribute values for an element are needed. The callback allows default attribute values to be provided when the default value is not specified in the DTD or schema or when a default value should be inherited from other elements. This callback is only used for Column View display, not during DITA document publishing.

This function returns either a count of the number of items added to *attrArr*, `0` if no items were added to *attrArr*, or `-1` if there was an error.

Arguments:

- *doc* — The identifier of the document containing the element for which you want to provide default values.
- *oid* — The object identifier (OID) of the element for which you want to provide default values.
- *attrArr* — An input/output associative array with the array key being the attribute names and the array values being the default values for the corresponding attribute.

  As a special case, when the callback is called to obtain default values to display for Column view, the first attribute name in *attrArr* will be the string `*ElementName*`. If its value is changed, then that value will replace the element name in the Column view **Outline** column.

  Note that attributes can have a legitimate value of the empty string, for example if the attribute is a CDATA attribute. To distinguish between an attribute that is not defaulted and an empty string value, the value of `*ATI_EMPTY_STRING*` should be returned for any attribute that has a legitimate value of the empty string. Any calling program must be prepared to accept and handle this special value.

  If multiple callbacks are registered, *attrArr* will accumulate the default values from each of the routines. Callback routines can overwrite values provided by previously called routines, but they should do so deliberately and with care. In general, separate callbacks will deal with different sets of attributes. For example, one callback might be registered to deal with the standard DITA elements and attributes, while another might be registered to deal with a set of namespaced elements or attributes. You can leave the default value for an attribute empty (null) when there is no default value for the attribute or when the particular routine does not know the default value for the attribute. The callback routine might be called with the names of attributes in *attrArr* that do not exist for the element represented by *oid* or for namespaced attributes, so the callback must be prepared to deal with this by returning a default value, a null string, or returning no value.
- *flag* — An optional set of option bit flags which default to zero. Allowed values are:
  - `0x0000` — Process normally, adding default values that are inherited from ancestors and application and processor default values.
  - `0x0001` — Process as with flag `0x0000`, except for DITA map documents. For DITA map documents, add default values that are inherited from ancestors and application and processor supplied default values for only the *scope* and *format* attributes.

- `0x0002` — Process as with flag `0x0000`, except for DITA map documents. For DITA map documents, do not add default values that are inherited from ancestors or any application or processor supplied default values for any attributes. (`0x0002` overrides `0x0001`.)

Other bits are unused and should be zero.

completeness_check Callback Type

Function prototype:
`function funcname (doc, logname)`

`completeness_check` is called when a completeness check is being performed on the specified document.

Arguments

- *doc* is the identifier of the source document.
- *logname* is the name of the event log being used for reporting the completeness checking. It is an empty string if the results are not being logged.

This callback can add reports to the event log. The callback should return the number of errors reported to the event log.

clone Callback Type

Function prototype:
`function funcname (doc, clone)`

`clone` is called when the specified document is cloned by the `doc_clone` function to create a new document. This action can not be cancelled by this callback.

Arguments

- *doc* is the identifier of the source document.
- *clone* is the identifier of the newly cloned document.

A document may be cloned during some publishing processes. This callback allows an action to be taken on the cloned document before further processing is performed.

conref_content Callback Type

Function prototype:
`function funcname (window, oid, reference)`

`conref_content` is called to resolve a content reference by a *conref* or a resolved *conkeyref* attribute during generated text processing.

Arguments

- *window* is the identifier of the window containing the document whose *conref* or *conkeyref* content is being resolved.
- *oid* is the object identifier (OID) of the element with the *conref* or *conkeyref* attribute value to be resolved.
- *reference* is the name of the attribute that is being resolved. Allowed values are `conref` and `conkeyref`. The default is `conref`.

The function must return the content to be inserted as an XML markup string, for example, returning the result from calling `oid_content` on an element in the referenced document. If it returns a null string, the default processing is done. Note that an empty string is also valid content to return. To return an empty string as content, return the value `*ATI_EMPTY_STRING*`.

Note

The `conref-content` callback type is used only for DITA document types.

context_changed Callback Type

Function prototype:
`function funcname (doc, oid, pos)`

`context_changed` is called when the caret has moved within the specified document to cause a change in context. This action can not be cancelled by this callback.

Arguments

- *doc* is the document containing the active cursor.
- *oid* is the OID containing the active cursor.
- *pos* is the number of characters into the OID where the cursor is located.

The location within the document specified by *oid* and *pos* must be a valid insertion position. If the cursor is currently inside generated text, this callback will be called with a position that is valid for element insertion. You can call the `in_context_list` function on page 384 to find out which elements are valid at a particular position.

context_error Callback Type

Function prototype:
`function funcname (doc, opcode, errcode, object)`

`context_error` is called when a context error is detected after certain edit operations, such as inserting tags or text.

`context_error` must return one of the following values:

- `-2` — Handled the error, the operation will return failure status.
- `-1` — Handled the error, the operation will return success status.
- `0` — Did not handle the error so normal error processing should be done. That is, the context error message will be displayed.

> **Note**
>
> The variable `main::ERROR` is set before `context_error` is called to hold the error message to be displayed to the user.

`context_error` will not be called if:

- an attempt to do a pending delete caused a context error
- an auto-inserted tag caused a context error

If pending delete succeeded, but an error occurred, `context_error` will be called after the delete. If `context_error` returns `0`, the delete will be undone, otherwise it will not be undone.

Arguments:

- *doc* is the identifier of the document in which the insertion was attempted and can be used to derive the cursor position and/or the selected region.
- *opcode* specifies the operation being attempted and currently is one of the following values:
  - `1` — insert tag
  - `2` — insert character or string
- *errcode* defines the type of error and is one of the following values:
  - `1` — selected region is not balanced (for inserting tag pairs only)
  - `2` — document is read-only
  - `3` — cursor is in a protected region
  - `4` — element is invalid at the cursor or at the beginning of the region if tag pairs are being inserted

> **Note**
>
> The error codes indicate the order of testing. For example, if the region is not balanced, then the test for a protected region has not yet been done. If in a protected region, the test for a context error has not yet been done.

- *object* is opcode specific:
  - for insert tag — it is the name of the tag for which insertion failed
  - for insert character or string — it is the character or string

copy Callback Type

Function prototype:
`function funcname (doc, buffername, op)`

`copy` is called before a copy operation is done in the Edit pane.

It is not called if the copy will request an external selection (that is, Arbortext Editor does not own the selection). This caveat does not apply to Windows platforms, because they do not support external selections.

A drag and drop action uses a special paste buffer named *_APT_DRAGDROP_PASTEBUF*. You can check *_APT_DRAGDROP_PASTEBUF* to tell if the copy operation is due to a drag and drop action.

Arguments

- *doc* is the identifier of the document in which the copy was attempted and can be used to derive the cursor position and/or the selected region.
- *buffername* is the name of the paste buffer that is the target of the copy. The standard paste buffer is named "`default`".
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.

- `-1` — Basic Arbortext Editor processing prevented.

create Callback Type

Function prototype:
`function funcname (doc)`

`create` is called whenever a new document is created by an `edit` or `new` command, by a call to `doc_open`, or when a file (external) entity is loaded.

Argument

doc is the identifier of the document created.

cut Callback Type

Function prototype:
`function funcname (doc, buffername, op)`

`cut` is called before a cut operation is done in the Edit window.

A drag and drop action uses a special paste buffer named *_APT_DRAGDROP_PASTEBUF*. You can check *_APT_DRAGDROP_PASTEBUF* to tell if the cut operation is due to a drag and drop action.

Arguments

- *doc* is the identifier of the document in which the cut was attempted and can be used to derive the cursor position and/or the selected region.
- *buffername* is the name of the paste buffer that is the target of the cut. The standard paste buffer is named "`default`".
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

delete Callback Type

Function prototype:
`function funcname (doc, op)`

`delete` is called before a delete operation occurs in the Edit window.

Arguments

- *doc* is the identifier of the document in which the delete was attempted and can be used to derive the cursor position or the selected region.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
    1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
        - `0` — Continue callback processing.
        - `-1` — Stop further callback processing.
    2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
        - `0` — Basic Arbortext Editor processing allowed.
        - `-1` — Basic Arbortext Editor processing prevented.

delete_region Callback Type

Function prototype:
`function funcname (doc, buffername, startoid, startpos, endoid, endpos, flags, op)`

`delete_region` is called before an attempt to delete a contiguous region of a document in an edit window. This callback can be used in place of the `delete`, `pending_delete`, and `cut` callbacks.

Arguments:

- *doc* — The identifier of the document in which the delete was attempted.
- *buffername* — The name of the paste buffer that is the target if the delete is a cut operation. The standard paste buffer is named `default`. If the delete is not a cut, then *buffername* will be an empty string.
- *startoid* — The object identifier representing the position of the beginning of the region to be deleted.
- *startpos* — The position from *startoid* (in characters) of the start of the region to be deleted.
- *endoid* — The object identifier representing the position of the end of the region to be deleted.

- *endpos* — The position from *endoid* (in characters) of the end of the region to be deleted.
- *flags* — The following flags can be set:
  - `0x01` — The selected region is the one to be deleted.
  - `0x02` — The table selection is to be deleted.
  - `0x04` — The delete is a pending delete of an insert operation.
  - `0x08` — The delete is not cancellable from the hook. (The return code from the callback will be ignored.) When set, the callback should be considered informational only and the callback should not attempt the delete itself.
- *op* — The function callback operation. It is generally called twice in succession
  1. First call: `op == 1`. Tests to verify the operation can be attempted. Return arguments are:
     - `0` — Proceed to the Execute step.
     - `-1` — Ignore callback. The code should continue as before.
  2. Second call: (occurs unless the process was stopped in the first step) `op == 2`. Instructs to continue and process the request. Return arguments are:
     - `0` — The code should still control processing.
     - `-1` — The code should be skipped (cancels the delete and the insert operation if the delete is a pending delete).
     - `1` — The delete should be skipped, but proceed with the insert operation (only applies for a pending delete).

delete_tag Callback Type

Function prototype:
`function funcname (doc, tagname, oid, op)`

`delete_tag` is called before a Delete Markup operation is done in the Edit window.

Arguments

- *doc* is the identifier of the document in which the Delete Markup was attempted and can be used to derive the cursor position and/or the selected region.
- *tagname* is the name of the tag that will be deleted (generally, to the left of the caret position).

> **Note**
>
> If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

- *oid* is the object identifier of the element to be deleted.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

destroy Callback Type

Function prototype:

```
function funcname (doc)
```

`destroy` is called whenever a document is unloaded from memory.

Argument

doc is the identifier of the document being destroyed.

enter_key Callback Type

Function prototype:

```
function funcname (doc, flags, op)
```

`enter_key` is called whenever the **ENTER** key is pressed. The default binding can be changed using this callback.

Arguments

- *doc* is the current document associated with this function.
- *flags* is a bit mask providing information about the following operation: `0x0001` — the **SHIFT** key is pressed.
- *op* is the standard `doc_add_callback` operation number (1 on the first call, 2 on the second).

This callback can be used to implement application-specific behavior for the **ENTER** key for a document or a document type, or for all documents if the callback is registered with `0` as the first parameter to `doc_add_callback`.

entity_notation Callback Type

Function prototype:
`function funcname (doc, entity, pubid, sysid, type, notation)`

`entity_notation` is invoked whenever an NDATA-type entity reference is inserted into the document. It is also called for each NDATA-type entity encountered when the document is initially opened. It is not called for NDATA-type entities assigned as attribute values.

Normally, Arbortext Editor treats NDATA entities as "graphic" entities. NDATA entities are treated specially only when they are assigned to a designated attribute of an element marked as a graphic tag. Arbortext Editor ignores graphic entities inserted into the document body (for example, using `insert("&ndataent;")`) because they have no effect on the document structure. This callback type allows the application writer to provide any special processing for such entities.

Arguments

- *doc* is the identifier of the current document.
- *entity* is the name of the entity inserted.
- *pubid* is the public identifier or a null string if no PUBLIC ID was specified in the entity declaration.
- *sysid* is the system identifier or a null string if no SYSTEM ID was specified in the entity declaration.
- *type* is the declared type of the entity and is always "NDATA".
- *notation* is the specified data notation.

The return value, if any, of the callback function is ignored.

entity_path Callback Type

Function prototype:
`function funcname (doc, entname, pubid, sysid, type, notation)`

`entity_path` is called to resolve an external entity and allows an application to override the default way of looking up an external entity path name. It must return the path name of a file to be opened by the standard I/O library functions `open` or `fopen` or the null string, if it is not able to associate a system path name with the entity. If a null string, the default processing is done.

If the path name returned is a file (that is, it does not contain any directory components), Arbortext Editor searches the document directory for the name if appropriate, and if a graphic entity, searches the path list specified by the environment variable *APTCATPATH* to locate the file.

Arguments

- *doc* is the identifier of the document associated with the entity path.
- *entname* is the name of the entity to be resolved.
- *pubid* is the public identifier or a null string if no PUBLIC ID was specified on the entity declaration.
- *sysid* is the system identifier or NULL if no SYSTEM ID was specified on the entity declaration.
- *type* is the declared type of the entity and is one of the strings "SUBDOC", "NDATA", "CDATA", "SDATA" or the null string if no type was declared (that is, an SGML text entity).
- *notation* is the specified data notation if entname is a graphic entity, or the null string otherwise.

Example

Here is an example function that implements the default entity path name processing:

```
function entpathhook(doc, entity, public, system, type)
{
   # if a PUBLIC id given, lookup that first and return
   # associated path if found
   if (public)
      {
      local path = public_id_path(public)
          if (path) { return path;}
      }
# otherwise, return SYSTEM if given, else ENTITY name
return system ? system : entity;
}
```

The body of this example function could be replaced by a call to the built-in function `entity_path` on page 345.

exclude_graphic_notation Callback Type

Function prototype:

```
function funcname (doc, notationname)
```

`exclude_graphic_notation` is called before a notation is added to the notation dropdown of the graphic entity dialog box. If `exclude_graphic_notation` returns `0`, the notation will be added to the notation dropdown. Otherwise, the notation will not be added to the notation dropdown.

exclude_tag Callback Type

Function prototype:

```
function funcname (doc, tagname)
```

`exclude_tag` is called before a tag is added to the insert markup panel. `exclude_tag` is also called when a menu is posted that contains tag-type menu items. In this case, if `exclude_tag` returns `0`, the tag menu item is enabled. Otherwise, the menu item is grayed out (made insensitive).

`exclude_tag` is only called for tags that are in context (that is, it may be inserted at the current cursor position.

Note

If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case-sensitive.

generate_id Callback Type

Function prototype:

```
function funcname (doc, oid, attrname)
```

`generate_id` is called to provide a generated ID to replace an ID that would otherwise be generated by Arbortext Editor. It returns the generated ID string or null, if no ID is being generated.

Arguments

- *doc* is the identifier of the document for which the ID is being generated. When an ID is being generated in a context that does not involve an open or existing document, the value of *doc* will be zero.
- *oid* is the object identifier of the element for which the ID is being generated. When an ID is being generated in a context that does not involve a particular element, the value of *oid* will be `oid_null()`.
- *attrname* is the name of the attribute for which the ID is being generated or a null string when the ID is not being generated for use as an attribute value.

Note

The `generate_id` callback type is currently used only for DITA document types.

include_path Callback Type

Function prototype:

```
function funcname (doc, attrs[])
```

`include_path` is called when an XML inclusion section of a document is initially expanded.

Arguments:

- *doc* — The identifier of the document containing the XML inclusion reference.
- *attrs[]* — An associative array containing the attributes of the XML inclusion tag. Any values changed by the callback will be used instead of the original values if the callback returns `-1`.

Also refer to the `entity_path` callback type on page 991.

Following is an example of implementing the `include_path` callback type.

```
function my_include_path_callback(doc, attrs[])
{
 local original_href = attrs['href'];
 local new_href = original_href;
 # possibly alter new_href here.
 if (original_href != new_href)
 {
 attrs['href'] = new_href;
 return -1; # force use of this new href
}
 return 0; # continue as normal
}
doc_add_callback(doc, 'include_path', 'my_include_path_callback')
```

insert_content Callback Type

Function prototype:

```
function funcname (doc, startoid, startpos, endoid, endpos)
```

`insert_content` is called by the `insert_string` command, and the `insert` command and function after a string that contains markup is successfully inserted. `insert_content` is also called by the `read` command after a successful insertion.

`insert_content` is not called for normal typing, inserting of symbols, or inserting text with the `insert` command or function.

If a pending delete occurs as a result of the insertion, then the pending_delete_after callback type on page 1005 is called instead of `insert_content`.

Arguments

- *doc* — identifier of the current document.
- *startoid* — the object identifier representing the position of the beginning of the inserted content.
- *startpos* — the position from *startoid* in characters of the start of the inserted content.
- *endoid* — the object identifier representing the position of the end of the inserted content.
- *endpos* — the position from *endoid* in characters of the end of the inserted content.

insert_entity Callback Type

Function prototype:
`function funcname (doc, entity, op)`

`insert_entity` is called by the `insert_entity` command before the entity reference is inserted. If the callback function inserts the reference itself, for example, using the `insert` function, it should return `-1`.

Arguments

- *doc* is the identifier of the current document.
- *entity* is the name of the entity reference to insert. The name is prefixed with the entity reference open character '&'.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

insert_include Callback Type

Function prototype:
```
function funcname (doc, uri, parse, op)
```

`insert_include` is called before an XML inclusion reference is inserted by the `insert_include` on page 660 command. If the callback function inserts the reference itself, it should return `-1`.

Arguments:

- *doc* — The document in which the XML inclusion reference is being inserted.
- *uri* — The resolved path to the resource (file) to be inserted.
- *parse* — The value of the *parse* attribute of the XML inclusion.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

Example
```
function my_insert_include_callback(doc, uri, parse, op)
{
}
doc_add_callback(doc, 'insert_include', 'my_insert_include_callback')
```

insert_include_path Callback Type

Function prototype:
```
function funcname (doc, attrs[])
```

`insert_include_path` is called before an XML inclusion reference is inserted by the `insert_include` on page 660 command. The callback can optionally modify the path value that will be stored on the *href* attribute value of the inserted XML inclusion tag.

Arguments:

- *doc* — The document in which the XML inclusion reference is being inserted.
- *attrs[]* — An associative array containing the attributes of the XML inclusion tag. If any values changed by the callback, `-1` is returned.

Example

```
function my_insert_include_path_callback(doc, attrs[])
{
 local original_href = attrs['href'];
 local new_href = original_href;
 # possibly alter new_href here.
 if (original_href != new_href)
 {
 attrs['href'] = new_href;
 return -1; # force use of this new href
}
 return 0; # continue as normal
}
doc_add_callback(doc, 'insert_include_path', 'my_insert_include_path_callback')
```

insert_tag Callback Type

Function prototype:

```
function funcname (doc, tagname, op)
```

`insert_tag` is called before the markup specified by *tagname* is inserted into a document, by either the `insert_tag` on page 663 command, the `insert_graphic` on page 659 command, or a markup menu item. If the function inserts the markup itself, for example, using the `insert_tag` on page 387 function, it should return `-1`.

Arguments

- *doc* is the identifier of the document associated with this function.
- *tagname* is the markup specified.

  If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, then the *tagname* parameter will be case sensitive.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.

  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.

- `-1` — Stop further callback processing.

2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
   - `0` — Basic Arbortext Editor processing allowed.
   - `-1` — Basic Arbortext Editor processing prevented.

Note

The `insert_graphic` on page 659 command also uses the `insert_tag` callback to bring up a file chooser dialog box to locate a graphic file to reference. The `insert_tag` callback then inserts the graphic tag with the appropriate graphic reference. If you add your own `insert_tag` callback after the standard callback performed by the `insert_graphic` command, then it gets called after the graphic tag is inserted. To call your own callback before the tag is inserted, you need to prepend your callback to the list of callbacks by following the example:

```
doc_add_callback(0, 'insert_tag', 'my_insert_tag', 'PREPEND')
```

Using the insert_tag callback with tables

It is a common ACL practice to use a custom insert_tag callback function to perform custom tag insertion (above and beyond what Arbortext Editor would do) and then instruct Arbortext Editor to skip the insert that it would have performed by default. When working with tables, such inserts would need to be carefully crafted to adhere to the table model you are using. If the customized insert is not supported by the table model, Arbortext Editor prohibits the insert.

A better solution is to use the `insert_tag_after` callback when you are within a table.

insert_tag_after Callback Type

Function prototype:

```
function funcname (doc, tagname, oid, op)
```

`insert_tag_after` is called after the element *tagname*, specified by *oid*, is inserted into the document.

Arguments:

- *doc* is the identifier of the document associated with this function.
- *tagname* is the name of the element specified.

> **Note**
> If the document type definition for your document instance includes the `NAMECASE GENERAL NO` setting, the *tagname* parameter will be case-sensitive.

- *oid* is the object identifier of the start tag that was inserted into the document.
- *op* is the function callback operation. *op* is called twice in succession (*op* = `1` and *op* = `2`) as is the convention with these callback types, but because the tag insertion has already occurred, it cannot cancel the operation.

insert_tag_auto Callback Type

Function prototype:

```
function funcname (doc, oids[], op)
```

The `insert_tag_auto` callback is called after tags are automatically inserted when context checking adds required tags, or if auto insertions are specified by the `InsertAroundToFix`, `InsertAutoNested`, and `InsertAutoHeading` categories in the `DCF` file for the document type.

This function is called after the document is valid and all the changes are applied by the `insert_tag` on page 663, `insert_string` on page 661, `newline` on page 687, `read` on page 698, `paste` on page 689, and `split` on page 715 commands and by the `insert` on page 384 and `insert_tag` on page 387 functions. This function is called before the `insert_tag_after` callback is called by the `insert_tag` command and function.

Arguments

- *doc* is the identifier of the document associated with this function.
- *oids* is an array of object identifiers of the start tags that were automatically inserted, in the order they were inserted into the document.
- *op* is the function callback operation. *op* is called twice in succession (*op* = `1` and *op* = `2`) as is the convention with these callback types, but because the tag insertion has already occurred, it cannot cancel the operation.

keybase_list_changed Callback Type

Function prototype:

```
function funcname (doc, path)
```

The `keybase_list_changed` callback is called any time a DITA document's `ditakeybaselist` option is updated.

Arguments

- *doc* is the ACL ID of the document for which the `ditakeybaselist` option is updated.
- *path* is the updated list of paths.

link Callback Type

Function prototype:
`function funcname (doc, oid, pos, op)`

`link` is called by the `link` command before the link action is taken, except for URI links. See the callback type `linkuri` on page 1002 for additional information.

Arguments

- *doc* is the identifier of the current document.
- *oid* is the object identifier representing the current mouse (if `link mouse`) or caret (if `link caret`) position.
- *pos* is the position from *oid* in characters of the mouse or cursor as returned by `oid_mouse_pos(oid)` or `oid_caret_pos(oid)`.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

The callback function must return `-1` if it performs the link action; otherwise it should return `0` to let Arbortext Editor complete its normal processing.

Note

By default, double-clicking the left mouse button is mapped to a function that executes a `link` command. If it returns failure (meaning there is no object to link to), double-click the current word. A link callback function must not interfere with this operation. (That is, if it returns `-1`, it must perform a link.)

linkto Callback Type

Function prototype:

```
function funcname (doc, target, type, data, op)
```

`linkto` is called by the `link` command before the link action is taken. It is only called when either the `-idref` or `-uri` argument is specified for the `link` command. This is the only callback for the `link` command if one of these two arguments is specified.

Arguments

- *doc* is the identifier of the current document.
- *target* is the target of the link, either an IDREF or a URI.
- *type* is the type of the *target*. When *type* is `0`, the *target* is an IDREF. When *type* is `1`, the *target* is a URI.
- *data* is the string specified by the `-data` argument the `link` command.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.

  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

The callback function must return `-1` if it performs the link action; otherwise it should return `0` to let Arbortext Editor complete its normal processing.

linkuri Callback Type

Function prototype:
`function funcname (doc, oid, uri, op)`

When linking to a URI, the `link` command calls `linkuri` before any link action is taken. This is unlike all other `link` actions, which call `link` hooks instead.

Arbortext Editor determines if a link is to a URI when the `LinkUriAttr` tag is configured in the `.dcf` file or in a `link:href` e-i-c characteristic (**Link URI/ID** in a style panel).

Arguments

- *doc* — Identifier of the current document.
- *oid* — Object identifier representing the current mouse (if `link mouse`) or caret (if `link caret`) position.
- *uri* — target URI.
- *op* — Function callback operation. It is called twice in succession:
  1. First call: `op == 1` confirms the operation is to be attempted. Return arg:
     - `0` — proceed to Execute step
     - `-1` — ignore callback, code should continue as before
  2. Second call: (occurs unless the process was stopped in the first step) `op == 2` meaning: "go ahead and process request now." Return arg:
     - `0` — code should still handle
     - `-1` — code should skip

The callback function must return `-1` if it performs the link action; otherwise it should return `0` to let Arbortext Editor complete its normal processing.

Note

By default, double-clicking the left mouse button is mapped to a function that executes a `link` command. A link callback function must not interfere with this operation. (That is, if it returns `-1`, it must perform a link.)

Note

The normal processing for URIs is to check whether or not the URI is an ID contained in the same document, possibly prefixed by the # sign. If it is, Arbortext Editor moves the cursor to that point. No link action occurs in other cases.

modify_tag Callback Type

Function prototype:

```
function funcname (doc, oid, win, op)
```

`modify_tag` is called before a **Modify Attributes** dialog box is opened to modify the attributes of an element. This allows an application to automatically set certain attributes before the dialog box is displayed or to prevent the dialog box from being displayed (by destroying it). `modify_tag` applies only to elements, and cannot be used to set attributes of processing instructions.

`modify_tag` is called before the dialog box fields are initialized, so that any changes made to attribute values by the callback will be reflected in the dialog box. If `modify_tag` returns `-1`, the **Modify Attributes** dialog box is not displayed. for example, if `modify_tag` does not want the tag to be modified or sets the attributes itself.

Arguments

- *doc* is the identifier of the document associated with this function.
- *oid* is the object identifier of the tag being modified.
- *win* is the window identifier of the **Modify Attributes** dialog box.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.

  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

paste Callback Type

Function prototype:
`function funcname (doc, buffername, after, op)`

The `paste` hook is called before a paste operation is done in the Edit window.

A drag and drop action uses a special paste buffer named *_APT_DRAGDROP_PASTEBUF*. You can check *_APT_DRAGDROP_PASTEBUF* to tell if the paste operation is due to a drag and drop action.

Arguments

- *doc* is the identifier of the document associated with this function.
- *buffername* is the name of the paste buffer being pasted.
- *after* determines whether the table object(s) in the paste buffer will be pasted before or after the cell, row, or column containing the text caret. If *after* is zero, the table object(s) in the paste buffer will be pasted before the cell, row, or column containing the text caret. If *after* is not zero, then the table object(s) in the paste buffer will be pasted after the cell, row, or column containing the text caret. If the hook will be performing the paste itself, it can use this parameter to determine whether the *-after* flag should be supplied to the `paste` command.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

pending_delete Callback Type

Function prototype:
`function funcname (doc, op)`

`pending_delete` is called before a pending delete operation occurs in the Edit pane.

Arguments:

- *doc* — The identifier of the document in which the delete was attempted. It can be used to derive the cursor position or the selected region.
- *op* — The function callback operation. It is generally called twice in succession
  1. First call: `op == 1`. Tests to verify the operation can be attempted. Return arguments are:
     - `0` — Proceed to the Execute step.
     - `-1` — Ignore callback. The code should continue as before.
  2. Second call: (occurs unless the process was stopped in the first step) `op == 2`. Instructs to continue and process the request. Return arguments are:
     - `0` — The code should still control processing.
     - `-1` — The code should be skipped (cancels the pending delete and the insert operation).
     - `1` — The delete should be skipped, but proceed with the insert operation.

pending_delete_after Callback Type

Function prototype:

```
function funcname (doc, startoid, startpos, endoid, endpos)
```

`pending_delete_after` is called after any successful insertion that followed a pending delete.

Arguments

- *doc* — the identifier of the current document.
- *startoid* — the object identifier representing the position of the beginning of the inserted content.
- *startpos* — the position from *startoid* in characters of the start of the inserted content.
- *endoid* — the object identifier representing the position of the end of the inserted content.
- *endpos* — the position from *endoid* in characters of the end of the inserted content.

print_panel Callback Type

Function prototype:

```
function funcname (doc, prtcmd, op)
```

`print_panel` is called when the user dismisses the **Print Editor View** dialog box.

Arguments

- *doc* is the identifier of the document associated with this function.
- *prtcmd* is a string giving the actual print command used to execute the print request.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

Notes

prtcmd is the null string if the panel was dismissed without the user pressing the **OK** button.

protect Callback Type

Function prototype:

```
function funcname (doc, opCode, errorCode)
```

`protect` is called when an operation is about to fail because of protection. The callback function may tell Arbortext Editor to perform the operation regardless. Currently, this callback is only called when delete operations fail due to a read-only protection error.

Arguments:

- *doc* — The identifier of the document in which the operation is being attempted.
- *opCode* — A code describing the operation. Currently this value is always `1` for a delete.
- *errorCode* — The protection error code. Currently it may be one of the following values:
  - `2` — The document is read-only.

- `6` — The document object is read-only.
- `9` — The file entity or XML inclusion is read-only.
- `12` — The location in the document is read-only.

Return arguments are:

- `0` — Proceed and let the operation fail due to the protection error.
- `-1` — Ignore the protection error and let the operation continue.

quick_attribute Callback Type

Function prototype:

```
function funcname (doc, oid, win, op)
```

`quick_attribute` is called before a **Quick Attribute** dialog box is opened to modify the attributes of an element. This allows an application to automatically set certain attributes before the dialog box is displayed or to prevent the dialog box from being displayed (by destroying it). `quick_attribute` applies only to elements, and cannot be used to set attributes of processing instructions.

`quick_attribute` is called before the dialog box fields are initialized, so that any changes made to attribute values by the callback will be reflected in the dialog box. If `quick_attribute` returns `-1`, the **Quick Attribute** dialog box is not displayed. for example, if `quick_attribute` does not want the tag to be modified or sets the attributes itself.

Arguments

- *doc* is the identifier of the document associated with this function.
- *oid* is the object identifier of the tag being modified.
- *win* is the window identifier of the **Quick Attribute** dialog box.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

reference_modify Callback Type

Function prototype:
`function funcname (doc, oid, prevvalarr)`

`reference_modify` is called whenever a reference attribute has been added or modified. It is not called when a reference attribute is deleted. When `reference_modify` is called, any additions or changes to an attribute have already been made in the document. The callback is called once per element, even if more than one reference attribute for an element is being added or changed. If one or more modifications are being made as part of a single operation that modifies several elements, the callback is called once for each element. Note that if changes to reference attribute values are made from a `reference_modify` callback, the callback is not called again for these changes.

Arguments

- *doc* is the identifier of the document containing the reference or references for which the value is being added or changed.
- *oid* is the object identifier of the element containing the reference or references for which the value is being added or changed.
- *prevvalarr* is an associative array with an entry for each reference attribute that is changed. The array is indexed by the reference attribute name. The values are the original values of the reference attributes before the change. These values can be used by the callback to restore a particular element and attribute to its previous value.

The return value of the callback function is ignored and should be zero.

Note

The `reference_modify` callback type is currently used only for DITA references.

Multiple `reference_modify` callbacks can be registered. Due to this, the callbacks need to be written to guard against the case where one callback deletes an element or attribute from the document, making some of the information (*oid*, attribute name, or previous value in *prevvalarr*) passed to later callbacks invalid or inconsistent.

reference_path Callback Type

Function prototype:
`function funcname (doc, oid, attrname, refvalue)`

`reference_path` is used to resolve references other than graphic references. It returns the possibly modified reference value or null, if there was an error. The returned reference value is usually an absolute or relative path name or a Logical ID on page 1036. If *refvalue* is to be used unchanged, its value should be returned.

Arguments

- *doc* is the identifier of the document containing the reference for which the value is being resolved. If *doc* is set to zero, the current document is used.
- *oid* is the object identifier of the element containing the reference for which the value is being resolved.
- *attrname* is the name of the attribute for which the value is being resolved.
- *refvalue* is the value of the attribute for which the value is being resolved.

Note

The `reference_path` callback type is currently used only for DITA references.

save Callback Type

Function prototype:

```
function funcname (doc, op)
```

`save` is called before a document is saved.

Arguments

- *doc* is the identifier of the document associated with this function.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

saveas Callback Type

Function prototype:
`function funcname (doc, filename, ro, savetype, encoding, op)`

`saveas` is called when a `save_as` command is performed before the document is saved. The **Save As** dialog box issues a `save_as` command when the user selects **OK** after choosing a new path name for the document.

Arguments

- *doc* is the identifier of the current document.
- *filename* is the path name entered from the command line or selected by the user from the **Save As** dialog box.
- *ro* is `1` if the `-readonly` option was specified on the `save_as` command or if **Read Only** was checked on the **Save As** dialog box.
- *savetype* is an integer that indicates any change in file type.
  - `0` — No change.
  - `1` — Saving the SGML document as an XML document.
  - `2` — Saving the XML document as an SGML document.
- *encoding* is a string. It is a null string if the operating system's encoding is used to save the file. Otherwise, it is a encoding name. Refer to the table below for a list of available encoding names.

Adobe-Standard-Encoding	ISO-8859-9
ISO-8859-1	EUC-JP
ISO-8859-1-Windows-3.1-Latin-1**	Shift_JIS
ISO-8859-2	Big5
ISO-8859-3	GB_2312-80
ISO-8859-4	KSC_5601
ISO-8859-5	UTF-8
ISO-8859-7	US-ASCII
ISO-8859-8	ISO-10646-UCS-2

- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op` == `1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.

- `-1` — Stop further callback processing.

2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
   - `0` — Basic Arbortext Editor processing allowed.
   - `-1` — Basic Arbortext Editor processing prevented.

If `saveas` needs to perform some action after the document is saved, it must execute the `save_as` command and then return `-1` to indicate that the command completed.

Example

```
function saveashook(doc, path, ro, savetype, encoding, op) {
   if (op == 1) {
      return 0; # proceed
   }
   if (ro) {
      save_as -readonly $path
   } else {
      save_as $path
   }
   if (status != 0) {
      # command failed
   }
   # do extra code here ...
   return -1; # we did the save
}
```

tbl_cell_clear Callback Type

Function prototype:

```
function funcname (doc, target, content , span, attrs, op)
```

`tbl_cell_clear` is called before the `tbl_cell_clear` on page 544 ACL function is executed. It may be called by the user from ACL or internally by the table editor user interface.

Arguments

- *doc* is the identifier of the document containing the *target*.
- *target* is the table object ID (toid) of the cell to be cleared.
- *content* is a boolean: one (`1`) if the cell's content is going to be discarded, zero (`0`) if the cell's content is not being cleared.
- *span* is a boolean: one (`1`) if the cell is going to be unspanned, zero (`0`) if the cell's spanned state is not being changed.

- *attrs* is a boolean: one (`1`) if the cell's attributes are going to be set to their default values, zero (`0`) if the cell's attributes are not being changed.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

Note

Unlike many other Arbortext Editor callbacks, the table editor does not expect the callback function to perform the operation itself and return `-1`. If the callback function returns `-1`, the table editor will display an error message.

tbl_cell_span Callback Type

Function prototype:

```
function funcname (doc, ulid, lrid, op)
```

`tbl_cell_span` is called whenever the `tbl_cell_span` on page 547 function is called.

Arguments

- *doc* is the identifier of the document containing *ulid* and *lrid*.
- *ulid* is the table object ID (toid) of the upper-left corner cell of the region to be spanned.
- *lrid* is the table object ID (toid) of the lower-right corner cell of the region to be spanned.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:

- `0` — Continue callback processing.
- `-1` — Stop further callback processing.

2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
   - `0` — Basic Arbortext Editor processing allowed.
   - `-1` — Basic Arbortext Editor processing prevented.

Note

`tbl_cell_span` is not called if a group of cells is spanned during the processing of a rectangle copy operation.

tbl_cell_unspan Callback Type

Function prototype:
`function funcname (doc, cellid, op)`

`tbl_cell_unspan` is called whenever the `tbl_cell_unspan` on page 547 function is called.

Arguments

- *doc* is the identifier of the document containing *cellid*.
- *cellid* is the table object ID (toid) of the master cell that contains the spanned region's content and attributes.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.

1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
   - `0` — Continue callback processing.
   - `-1` — Stop further callback processing.
2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
   - `0` — Basic Arbortext Editor processing allowed.
   - `-1` — Basic Arbortext Editor processing prevented.

Note

`tbl_cell_unspan` is not called if a region is unspanned as the result of a rectangle copy operation.

tbl_grid_focus Callback Type

Function prototype:
`function funcname (doc, gridid, window, action)`

`tbl_grid_focus` is called whenever a table grid becomes active or inactive in the Edit window.

Arguments

- *doc* is the identifier of the document containing *gridid*.
- *gridid* is the table object ID (toid) of the table grid.
- *window* is the identifier of the window.
- *action* is a boolean: one (`1`) if the grid is becoming active, zero (`0`) if the grid is becoming inactive.

tbl_insert Callback Type

Function prototype:
`function funcname (doc, oid, pos, rows, cols, tmid, tag, op)`

`tbl_insert` is called before a table is inserted into the document.

Arguments

- *doc* is the identifier of the document containing *oid*.
- *oid* and *pos* indicate where the table is being inserted.
- *rows* and *cols* indicate the number of rows and columns (respectively) within the table that is being inserted.
- *tmid* is the table model ID of the table model that will manage the table being inserted.
- *tag* is name of the wrapper tag for the table being inserted.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.

- `-1` — Stop further callback processing.

2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
   - `0` — Basic Arbortext Editor processing allowed.
   - `-1` — Basic Arbortext Editor processing prevented.

Note

`tbl_insert` will not be executed if a table is inserted as the result of a copy from a paste buffer into a document.

tbl_insert_after Callback Type

Function prototype:

```
function funcname (doc, setid)
```

`tbl_insert_after` is called after a table is inserted into the document.

Arguments

- *doc* is the identifier of the document containing *setid*.
- *setid* is the table object ID (toid) of the TSet containing the table.

tbl_model_prompt Callback Type

Function prototype:

```
function funcname (doc, tmIDs[])
```

The `tbl_model_prompt` callback is called when users try to insert a table in their document, and the cursor is in a location in which it is valid to insert more than one table model.

Arguments:

- *doc* is the identifier of the document in which the table will be inserted.
- *tmIDs[]* is the array of valid table model IDs.

Return arguments:

- `-1` — Cancels the table insertion.
- `0` — Displays the prompt if `set prompttablemodel` on page 878 is `on`,

or automatically inserts the primary table model if `set prompttablemodel` is `off`.

- Non-zero integer — A one-based index value into the *tmIDs* array that indicates the table model to be inserted.

tbl_obj_add Callback Type

Function prototype:
`function funcname (doc, before, after, op)`

`tbl_obj_add` is called before a Grid, Column, or Row is inserted into an existing table.

Arguments

- *doc* is the identifier of the document containing *before* and *after*.
- *before* is the table object ID (toid) of the grid, column, or row after which the new grid, column, or row will be inserted.
- *after* is the table object ID (toid) of the grid, column, or row before which the new grid, column, or row will be inserted.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

Either the *before* or *after* arguments, but not both, may be `h::tblNullToid` if the new object will be inserted as the first or last grid, row, or column.

tbl_obj_add_after Callback Type

Function prototype:
`function funcname (doc, toid)`

`tbl_obj_add_after` is called after a grid, column, or row is inserted into an existing table.

Arguments

- *doc* is the identifier of the document containing *toid*.
- *toid* is the table object ID (toid) of the newly inserted grid, row, or column.

tbl_obj_attr_modifiable Callback Type

Function prototype:
`function funcname (doc, target, attr)`

`tbl_obj_attr_modifiable` is called by the table editor user interface to ask permission to modify a table object attribute.

Arguments

- *doc* is the identifier of the document containing *target*.
- *target* is the table object ID (toid) of the table object whose attribute is to be modified.
- *attr* is the name of the attribute to be modified. This callback only supports two attributes: `TARGET_HEIGHT` (rows) and `COLWIDTH` (columns).

Returns

`tbl_obj_attr_modifiable` may return:

- `0`: modification is permitted
- `-1`: modification is forbidden

A return of `-1` restricts modification by the row and column rulers, and also by the **Table Properties** dialog box. However, this callback does not let you restrict changes to the `TARGET_HEIGHT` and `COLWIDTH` attributes in other ways, such as using a direct ACL function call. It also does not let you restrict changes to other attributes.

Note

If the `set tablecolumnresizable` option is set to `off`, columns cannot be resized even if the `tbl_obj_attr_modifiable` callback allows modification.

This callback does not let you forbid changes to these attributes in other ways, for example, by a direct ACL function call.

tbl_obj_attr_set Callback Type

Function prototype:
`function funcname (doc, target, attr, old, new, op)`

`tbl_obj_attr_set` is called before a table object attribute is modified.

Arguments

- *doc* is the identifier of the document containing *target*.
- *target* is the table object ID (toid) of the table object whose attribute is to be modified.
- *attr* is the name of the attribute to be modified.
- *old* is the current attribute value.
- *new* is the intended attribute value.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

tbl_obj_delete Callback Type

Function prototype:

```
function funcname (doc, target, op)
```

`tbl_obj_delete` is called before a table, grid, column, or row is deleted.

Arguments

- *doc* is the identifier of the document containing *target*.
- *target* is the table object ID (toid) of the grid, column or row being deleted.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:

- `0` — Basic Arbortext Editor processing allowed.
- `-1` — Basic Arbortext Editor processing prevented.

Note

`tbl_obj_delete` is called only once per delete operation. For example, if a table is deleted, `tbl_obj_delete` will not be called for each grid, column, or row within the table.

tbl_recognize Callback Type

Function prototype:
`function funcname (doc, setid)`

`tbl_recognize` is called whenever the table editor notices that a document contains a table. This may occur as a document is opened, during the copying of data to or from a paste buffer, or when manipulation of a document's tags using ACL causes the a collection of tags to match a table template pattern.

Arguments

- *doc* is the identifier of the document containing *setid*.
- *setid* is the table object ID (toid) of the TSet containing the table.

tbl_rectangle_copy Callback Type

Function prototype:
`function funcname (doc, source_ul, source_lr, target_ul, target_lr, op)`

`tbl_rectangle_copy` is called before a rectangle of cells is copied from one table to another. Usually the copy takes place between a table in a paste buffer and a table in a document in an Edit window.

Arguments

- *doc* is the identifier of the document containing *target_ul* and *target_lr*.
- *source_ul* is the table object ID (toid) of the upper-left corner cell in the region from which the copy is taking place.
- *source_lr* is the table object ID (toid) of the lower-right corner cell in the region from which the copy is taking place.
- *target_ul* is the table object ID (toid) of the upper-left corner cell in the region where the table objects are being inserted.

- *target_lr* is the table object ID (toid) of the lower-right corner cell in the region where the table objects are being inserted.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

tbl_rectangle_copy_after Callback Type

Function prototype:
`function funcname (doc, source_ul, source_lr, target_ul, target_lr)`

`tbl_rectangle_copy_after` is called after a successful rectangle-to-rectangle copy.

Arguments

- *doc* is the identifier of the document containing *target_ul* and *target_lr*.
- *source_ul* is the table object ID (toid) of the upper-left corner cell in the region from which the copy is taking place.
- *source_lr* is the table object ID (toid) of the lower-right corner cell in the region from which the copy is taking place.
- *target_ul* is the table object ID (toid) of the upper-left corner cell in the region where the table objects are being inserted.
- *target_lr* is the table object ID (toid) of the lower-right corner cell in the region where the table objects are being inserted.

tbl_rectangle_dragable Callback Type

Function prototype:
`function funcname (doc, source_ul, source_lr, op)`

`tbl_rectangle_dragable` is called by the table editor user interface before a drag-and-drop operation begins. It lets you grant or deny the user interface permission to initiate a drag and drop operation.

Arguments

- *doc* is the identifier of the document containing *source_ul* and *source_lr*.
- *source_ul* is the table object ID (toid) of the upper-left corner cell in the region to be dragged.
- *source_lr* is the table object ID (toid) of the lower-right corner cell in the region to be dragged.
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.
  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

Returns

`tbl_rectangle_dragable` may return:

- `0`: the drag operation should proceed
- `-1`: the drag operation is forbidden

tooltip Callback Type

Function prototype:

```
function funcname (doc, oid, win, str)
```

`tooltip` is called when a tooltip is about to be raised over a Arbortext Editor window and allows the tooltip to be suppressed or changed.

Arguments

- *doc* the identifier of the document containing the element under the cursor.
- *oid* is the object identifier of the element under the cursor.
- *win* is the window identifier over which the tooltip is to be raised.
- *str* is the text that will be shown in the tooltip.

If an empty string is returned, the tooltip will not be raised. Otherwise, the text returned will be shown as the tooltip.

undo Callback Type

Function prototype:
`function funcname (doc, detail, startoid, startpos, endoid, endpos)`

`undo` is called after any undo or redo operation completes. The *detail* argument indicates the reason for the callback. The last four arguments give the range of the document affected by the undo operation.

Arguments

- *doc* — The identifier of the current document.
- *detail* — A code indicating the type of undo operation as follows:
  - 1 — The undo command.
  - 2 — The undo triggered by Arbortext Editor as the result of context errors.
  - 3 — The redo command.
- *startoid* — The object identifier representing the beginning position of the range affected by the undo operation.
- *startpos* — The position from *startoid* (in characters) to the beginning of the range affected by the undo.
- *endoid* — The object identifier representing the end position of the range affected by the undo.
- *endpos* — The position from *endoid* (in characters) to the end of the range affected by the undo.

doc_remove_callback

`doc_remove_callback` `(doc, cbtype, callback)`

Removes *callback* from the list of functions called when a document-level event occurs. If *doc* is invalid, or *callback* is not on the callback list for the document, $ERROR is set and `0` is returned. If successful, the function returns the unique document identifier.

The *doc* parameter is a document identifier. The *cbtype* parameter needs to be a valid callback type (valid callbacks are listed under the `doc_add_callback` function). The *callback* parameter specifies the actual callback to remove.

Example
```
$ret = doc_remove_callback($doc, "create", \
       "main::DocCallback")
```

Refer to Callback Functions introduction on page 112 for helpful overview information.

session_add_callback

session_add_callback (*cbtype*, *callback*[, *"PREPEND"*])

Appends *callback* to the list of callback functions to be called when desired session-level event occurs. If *"PREPEND"* is specified, the callback is prepended to the list of callback functions. Functions are called in the order that they appear in the list.

The *cbtype* parameter needs to be a valid callback type. If *cbtype* is invalid, the function returns 0. If it's successful, the function returns 1.

The *callback* parameter specifies the actual callback to add. The *"PREPEND"* flag specifies that the callback should be prepended, not appended.

Note

If *callback* is already on the callback list for the session, the list is shuffled to place the callback into the desired position; the callback is not added to the list a second time.

Refer to Callback Functions introduction on page 112 for helpful overview information.

Refer to the following valid session callback types.

`drop_file` on page 1023
`drop_file_over` on page 1024
`quit` on page 1025

drop_file

Function prototype:

```
function funcname(path, num, count, flags)
```

`drop_file` is called when a drop operation involving one or more files is performed. The default action can be prevented by returning `-1` from this callback. For example, if the callback opens or inserts the file(s).

Arguments:

- *path* is the path name of the file being dropped.
- *num* is the index number of the file being dropped, starting with `1`.
- *count* is the number of files being dropped by the operation.
- *flags* is bit mask giving information about the drop operation defined below.

If more than one file is being dropped, the callback is called multiple times with the *num* parameter increased incrementally on each call, until *count* calls have been made. If the callback returns `-1` on any call during a multiple-file drop, then the file associated with that callback is not processed and the next file is called.

The *flags* parameter may have any of the following bits set:

- `0x0001` - the **SHIFT** key is pressed.
- `0x0002` - the **CTRL** key is pressed.
- `0x0004` - the path specifies a graphic file.
- `0x0008` - the **ALT** key is pressed.

By default, the **CTRL** key modifier determines whether a file is opened or inserted. A graphic file (indicated by bit `0x0004` being set) is always inserted.

Note

This callback is only available on the Windows platform.

drop_file_over

Function prototype:

```
function funcname(op, count, flags)
```

`drop_file_over` is called to receive more information on the progress of a drag and drop operation and to provide feedback that will cause different mouse pointers, and possibly a drop cursor, to be displayed or not. The callback might be called multiple times (possibly hundreds of times) for a single drag and drop operation as the mouse pointer moves over the window, so should be written as efficiently as possible.

The drop file operation ends with either a call to the `drop_file` callback or a final call to the `drop_file_over` callback indicating that the drag and drop operation has been cancelled or is no longer over the Arbortext Editor window. If multiple `drop_file_over` callbacks are registered, the first callback that does not return `-1` determines what action will be taken. Any remaining callbacks will not be called until the mouse moves.

Arguments:

- *op* is an operation code that has the following possible values:
  - `1` — Indicates that a drag and drop operation is over the Arbortext Editor window.
  - `2` — Indicates that a previously started drag and drop operation is no longer over the Arbortext Editor window. This value is only made if the

drag and drop operation is canceled or the mouse pointer leaves the Arbortext Editor window.

- *count* is the number of files being dropped by the operation.
- *flags* is a bit mask giving the following information about the drop operation:
  - `0x0001` — The **Shift** key is pressed.
  - `0x0002` — The **Ctrl** key is pressed.
  - `0x0004` — The path specifies a graphic file.
  - `0x0008` — The **Alt** key is pressed.

The function returns one of the following codes:

- `-1` — Indicates processing should continue.

  That is, the next `drop_file_over` callback is called or default system processing is used to determine what mouse pointer and drop cursor to display.
- `0` — Indicates that the request is not allowed at this point.

  In this case, the `drop_file` callback, if any, will not be called.
- `1` — Indicates that the request is allowed and will result in a file open operation that is independent of the mouse position at the time of the drop.
- `2` — Indicates that a drop at this point will result in an insertion and that a drop insertion cursor should be drawn to show the point of insertion.

Note

This callback is only available on the Windows platform.

quit

Function prototype:

```
function funcname (code, system)
```

This `quit` callback is passed the argument *code*, which indicates how the application is being exited.

Arguments

- *funcname* is the name for the user-defined function called to be invoked whenever the editing session is terminated. This is invoked before any of the built-in save checking is done. If the `quit` callback returns a value of `-1`, no further callbacks will be called and the quit will be canceled. All other returned values are ignored. If the callback is already on the callback list for

the session, the list is shuffled to place the callback into the desired position. The callback is not added to the list a second time.

- *code* is one of the following values:
  - `0` — prompt about any unsaved changes, that is, quit
  - `1` — save all modified documents without prompting, that is, exit
  - `2` — do not prompt about unsaved changes and quit without saving modified documents, that is, `quit ok`
- *system* is one of the following values:
  - `0` — indicates a normal application exit.
  - `1` — indicates an application exit due to a system shutdown (due to a log off, restart, and so forth).

session_remove_callback

`session_remove_callback``(`*`cbtype`*`, `*`callback`*`)`

Removes *callback* from the list of functions called when a session-level event occurs. The *callback* parameter specifies the callback to remove.

The *cbtype* parameter needs to be a valid callback type (valid callbacks are listed under the `session_add_callback` function). If *callback* is not on the callback list for the session or if *cbtype* is invalid, the function returns 0. If it's successful, the function returns 1.

Example

```
$ret = session_remove_callback("quit", main:: DocCallback)
```

timer_add_callback

`timer_add_callback` `(`*`csecs`*`, `*`callback`*`[, `*`data`*`])`

This function creates an interval timer and returns an identifier for it. *csecs* is the time interval in centiseconds (hundredths of a second). *callback* is the name of the function to be called when the interval timer expires. *data* is an optional user data value that is passed to the callback.

The callback function must be declared as:

```
function callback([data])
```

and must return either `0` to cancel the timer or `1` to reenable it.

Due to the overhead of executing Arbortext Command Language code, time interval values of less than one-tenth of a second are not recommended.

In the example that follows, the `timer_add_callback` function displays the current time on the status bar (which is updated once per second for a minute).

```
global nticks=0
function update_clock(win)
{
   if (++nticks == 60) {
      window_set(win, 'message', '')
      return 0; # cancel timer
   }
      window_set(win, 'message', substr(time_date(), 12, 8))
      return 1; # reenable timer
}
timer_add_callback(100, 'update_clock', doc_window())
```

timer_remove_callback

timer_remove_callback(*id*)

This function cancels the interval timer specified by *id*, which is an identifier returned by a previous call to `timer_add_callback`. Note that the timer can also be cancelled in the callback function by returning `0`.

userule_add_callback

userule_add_callback (*docid*, *userule*, *userule_func*)

This function registers userules for document types. `userule_add_callback` is called when the specified userule(s) and document type(s) need processing. `userule_add_callback` returns `0` if successful. The function returns `-1` on registration failure.

The current registration of (*docid*, *userule*, *userule_func*) overrides any existing registration.

- *docid* — The document identi□er of a document using the document type. Setting *docid* to `0` registers the userule for all document types.
- *userule* — The userule number. Setting *userule* to `0` registers all userules for the document type. Setting *userule* to a value greater than 2 specifies a particular userule.
- *userule_func* — The userule function.

userule_remove_callback

userule_remove_callback(*docid*, *userule*, *userule_func*)

This function unregisters userules for document types. `userule_remove_callback` returns `0` if successful. The function returns `-1` on failure.

- *docid* — The document identi□er of a document using the document type. Setting *docid* to `0` unregisters the userule for all document types.
- *userule* — The userule number. Setting *userule* to `0` unregisters all userules for the document type. Setting *userule* to a value greater than 2 specifies a particular userule.
- *userule_func* — The userule function.

window_add_callback

window_add_callback (*window*, *cbtype*, *callback*[, *"PREPEND"*])

Appends *callback* to the list of callback functions to be called when the value of a dialog box item within *window* changes, or when another window-level event occurs. If *"PREPEND"* is specified, the callback is prepended to the list of callback functions instead. Functions are called in the order that they appear in the list. If *window* is invalid, $ERROR is set and `0` is returned. If successful, the function returns the unique window identifier.

The *window* parameter is a window identifier. A valid window identifier must be specified. (A value of `0` specifies “all windows”.) The *cbtype* parameter needs to be a valid callback type. (Refer to the following list of *Valid window callback types*.) The *callback* parameter specifies the actual callback to add. The *"PREPEND"* flag specifies that the callback should be prepended, not appended.

Note

If *callback* is already on the callback list for the window, the list is shuffled to place the callback into the desired position; the callback is not added to the list a second time.

Note

Arbortext Editor supports a single callback of each type per window.

Example

```
$ret = window_add_callback (window, cbtype, callback)
```

Refer to `window_set` on page 597 and Callback Functions introduction on page 112 for related information.

Refer to the following valid window callback types.

create on page 1029
destroy on page 1029
notify on page 1029
quit on page 1029

create Callback Type

create is invoked whenever a window is created either by a call to the window_create function or a call internally by Arbortext Editor to create a window.

destroy Callback Type

destroy is called whenever a window is deleted. The global destroy hook is called after any local destroyCallback function is set on the window.

notify Callback Type

notify is used when you add a callback to a XUI window. It has the following form:

```
callback(windlgitemeventdataappdata)
```

Where:

- *win* — The identifier of the window to be removed.
- *dlgitem* — The value of the control's id attribute.
- *event* — The type of event. It can be one of the following strings:
  - WINDOW_LOAD — When the window is first loaded.
  - WINDOW_CLOSED — When the window is closed.
  - WINDOW_FOCUSED — When the window is activated.
  - WINDOW_UNFOCUSED — When the window is deactivated.
  - WINDOW_DOCK_CHANGED — When the docking state of the dialog is changed. For this event only, *data* holds the old docking state. For example, "top", "right", "none", and so on.
- *data* — User data value that is passed to the callback.
- *appdata* — Specifies a value for later reference.

quit Callback Type

quit is called when a window is dismissed.

Example

```
function funcname (win, code, op=2)
```

The window identifier, *win*, can be obtained by calling `window_id`. The argument *code* is passed, indicating how the application is being exited.

Arguments

- *win* is the identifier of the window to be removed.
- *code* is one of the following values:
  - `0` — prompt about any unsaved changes, that is, `quit`
  - `1` — save all modified documents without prompting, that is, `exit`
  - `2` — do not prompt about unsaved changes and quit without saving modified documents, that is, `quit ok`
- `op` is the function callback operation. Callbacks are called twice in succession with `op` specifying the stage of callback operation.
  1. `op == 1` first call — The returned argument specifies whether the execution should continue or be stopped:
     - `0` — Continue callback processing.
     - `-1` — Stop further callback processing.

     Note

     The `op` argument is only passed if the callback function was added as a global callback (that is, registered with the *win* argument to `window_add_callback` set to `0`. For a local callback, the `op` argument is not passed (since the callback is actually implemented using the quit callback attribute of `window_set`). Because of this, the callback function must assign a default value of 2 to the `op` parameter (if it will be used as a local callback).

  2. `op == 2` second call — Occurs unless the processing was stopped during the first call. The returned argument allows or prevents basic Arbortext Editor processing after all callbacks have been called:
     - `0` — Basic Arbortext Editor processing allowed.
     - `-1` — Basic Arbortext Editor processing prevented.

window_remove_callback

`window_remove_callback``(`*`window`*`, `*`cbtype`*`, `*`callback`*`)`

Removes *callback* from the list of functions called when the value of a dialog item within *window* changes value, or when another window-level event occurs. If *window* is invalid, or *callback* is not on the callback list for the window, $ERROR is set and 0 is returned. If successful, the function returns the unique window identifier.

The *window* parameter is a window identifier. The *cbtype* parameter needs to be a valid callback type (valid callbacks are listed under the `window_add_callback` function). The *callback* parameter specifies the actual callback to remove.

Example

```
$ret = window_remove_callback($win, "create", main:: WindowCallback)
```

9

Repository API

Introduction to the Repository API

The Repository API allows Arbortext Editor authors to work with virtual documents made up of separate document objects that are stored and controlled independently. This makes it easy for Arbortext Editor users to share and reuse pieces of tagged text (SGML, XML, or HTML) in multiple documents. These individual components can be file entities or externally managed virtual documents. File entities and document objects may be edited directly within their parent document(s).

Stored document objects are accessed via the Repository API. An additional set of Arbortext Command Language (ACL) functions are provided to handle the basics of object manipulation such as browsing, loading, and saving. The Repository API is generalized, and can be applied to a wide variety of document repositories. An Adapter is designed to handle the details of the interface between Arbortext Editor's Repository API and a specific document repository.

Persistent Object Identifiers (POIDs)

A document object is identified by a Persistent Object ID, or POID. A POID is a string that uniquely identifies a specific revision of a specific object in a specific document database (or docbase). The precise format is left to the developers of a particular repository adapter, but adapter writers should use a standard format such as an Internet URL. Furthermore, all POIDs for a specific repository should have a unique signature at the beginning called the poid prefix. For URL-style POIDs, this would be encoded in the **protocol** field.

For example, consider a hypothetical repository called SGMBASE. A URL-style POID (assuming that “SGMBASE” does not conflict with any Internet standard protocol names and it has been registered as unique with Arbortext) would have the signature `x-sgmbase://` and an example POID would be `x-sgmbase://docbasename-65AF54AAF-01002` , where *docbasename* is the name of the specific docbase containing the object.

Logical IDs

The Logical ID is a unique identifier of an object that is used with file entity references. The **Logical ID** is similar to the POID, but can indicate the binding from the object to the object's parent.

Logical IDs vary depending on the repository being used. Refer to the documentation for your specific adapter for a description of its logical ID syntax. Typically, the logical ID contains the following information:

- The name of the repository containing the object.
- An identifier generated by the adapter that uniquely identifies the object.
- If the link was dynamic (that is, always uses the most recent version), the Logical ID will end with an adapter-specific label indicating a dynamic binding. If the link was static (that is, to a specific version), the Logical ID will end with the object's version label as generated by the repository server.

Customizing Entity Names for Objects

When you create a new entity object by inserting an object from the browser, a function in the Repository API automatically assigns that entity a unique name (for example, `E1340`). If you wish, you can write your own function for creating entity names, and substitute it in place of the default function.

To substitute your own entity naming function:

Create an ACL function that creates your desired unique entity names. Observe the following guidelines when creating this function.

- Your function should accept the following arguments:
  - *sesshdl* — ACL session handle for the current Arbortext Editor/adapter connection.
  - *poid* — Persistent Object ID for the entity to be created.
  - *is_graphic* — Boolean value indicating whether the new entity object is a graphic. One (1) indicates a graphic entity; zero (0) indicates a non-graphic entity.
  - *doc* — ACL document ID for the top level document object.
- Your function should return a unique entity name in the form `&`*`entname`* where *entname* is the name for the entity, and the `&` is a required prefix character. If a given document contains two different entities that possess the same entity name, the first of the two entities declared will appear in place of both entity references. If you are unsure if your function will always return unique names, you should implement the `entitydeclconflicthook` and create a function to handle the case when duplicate names are created.
- The ACL package that contains your entity naming function must be loaded before a user can connect to the repository. Since it is common for documents of different document type definitions (DTDs) to reside in unique document databases, it is typical that the entity naming function be added to the *`doctype`*`.acl` file, where *doctype* is the file system name for the document type you are using. This would guarantee that the entity naming function would be loaded whenever a file of that document type is loaded/created.

In the file where you create your entity naming function, set the global variable that defines the name of the entity naming function. Add the following line to the file containing your function:

```
$dms::entity_naming_function = "my_function_name
"
```

In the example above, *my_function_name* is the name of your desired entity naming function. Make sure that the function name is surrounded by double-quotation marks (").

Connecting to Document Management Systems

The Repository API defines a session that is an abstraction of a connection to a particular document database in a particular document repository by an authorized user. A connected session is a prerequisite to performing document management or browsing operations on objects in the repository. The underlying repository will require the Arbortext Editor user to log in with a user ID and password before authorization to access document objects is granted.

Document Object Management

The Repository API is used by Arbortext Editor to access common document management services provided by the underlying repository software.

The supported operations include:

- Access Control—The Repository API does not provide any access control scheme; it simply provides the hooks for a string describing a document's access privileges to be passed back and forth. The actual granting or denial of access to a particular document by a particular user is the responsibility of the repository. (Arbortext Editor does not support any modifications of repository permissions from Arbortext Editor; access control specifications must be made in the repository directly.)
- Revision Control —The Repository API revision control system scheme is general enough to encompass most revision control schemes. It is inspired by RCS, but has the major difference that it creates a persistent working version of an object as soon as a parent revision is locked. This frees you from the necessity of managing a local version of a locked object until it is checked in. Instead, the working version of an object is immediately created, all subsequent **save** operations will update the working version in place, then the **Object check in** operation makes the working version the latest version in the revision list.

Document objects may be checked out and in from Arbortext Editor. When a document object is checked out, a new revision is created in the database. Subsequent saves will update this new revision; and the object may be checked in to make the new revision permanent, or unlocked to remove it. Each revision of a document object has a unique POID, so the Arbortext Editor user can access any version of a document object.

- Document I/O and management—Document objects may be loaded and saved from within Arbortext Editor. Document objects may also be copied or deleted with the Repository API. Repository API routines do not attempt to provide a full range of document object manipulation and administration facilities; those tasks should be handled by the utilities in the underlying repository.
- Accessing Document Object Metadata—The Repository API requires a base set of metadata fields. Additional metadata fields may be defined by customers or Application Developers to the extent that is feasible on a given platform and for a particular Arbortext Editor application. The Repository API and the Arbortext Editor ACL allow a passthrough between a custom user interface and a custom database to store and retrieve any value of any metadata field.

Browsing and Searching for Document Objects

The Repository API supports browsing through a set of document objects. You may browse through:

- all objects contained in a folder or virtual document
- all versions of a specific object
- the set of objects matching a search criteria
- the set of objects that contain a given document object

Error Message Handling in the Repository API

Most non-Repository API ACL functions write error messages to the global variable `$ERROR` . The functions in the Repository API do not behave the same way. If a Repository API function returns an error state (usually `-1`), the developer should include code to obtain the error text from `main::ioerr` variable. If the `main::ioerr` variable contains a value of `-183`, then the developer should invoke the `sess_err_message` function to retrieve the text for the session error that the Repository API function generated.

A sample of such code appears below.

```
# The dobj_lock() function is used for this example.
#
```

```
$ret = dobj_lock(..)
#
# Test for errors, values less than zero are errors.
#
if ($ret <=0)
{
#
# Retrieve the error code from main::ioerr.
#
 local err_code = main::ioerr;
#
# If the error code is -183, retrieve the session error
# text using the sess_err_message function. Otherwise,
# retrieve the error text from the dms::io_err_msg array
# using the error code as the index into the array.
#
 if (err_code == -183)
 {
 err_message = sess_err_message(sesshdl);
}
 else
 {
 err_message = dms::io_err_msg[err_code]
}
}
```

Repository API Functions — Overview

The Repository API functions facilitate communications between Arbortext Editor and a variety of document repositories. You can write corresponding Repository API callbacks (see Repository API Callback Functions on page 1074) for some of these functions.

burst_multiple

burst_multiple (*filepath*[, *logfile*[, *flags*]])

This function opens each file specified by *filepath* individually, bursts them according to the current bursting configuration, and stores the resulting document objects to the connected document repository. The *filepath* can include wildcard characters.

If your burst configuration is set to automatically burst existing file entity references, your *filepath* should exclude file entities; they will be burst when the parent document is burst.

The optional *logfile* parameter specifies where bursting information will be written. If this parameter is not provided, the log information is written to the directory specified by the *TMPDIR* environment variable in a file called `burst`

NN`.log`, where *NN* is a number. If the *TMPDIR* environment variable is not set, the function attempts to write the logging information to the `c:\temp` directory on Windows or the `/tmp` directory on UNIX.

The optional *flags* parameter specifies overrides to default bursting options. If omitted, the function uses the default bursting options. The *flags* parameter can use the following bits:

- `0x0001` — enable full text indexing on the top most object
- `0x0002` — import file entities
- `0x0004` — don't import file entities
- `0x0008` — import graphic files
- `0x0010` — don't import graphic files
- `0x0020` — burst on element boundaries
- `0x0040` — don't burst on element boundaries
- `0x0080` — use the filename for the topmost object name
- `0x0100` — don't use the filename for the topmost object name
- `0x1000` — use location rules for child objects even when `useroverride="on"` in the burst configuration file

The negative settings override session defaults.

Note

A session must be established with the repository before you can use this function.

Examples:

```
burst_multiple("c:\\docs\\*.sgm")
burst_multiple("/docs/*.xml","/docs/burst.log",0x0010)
```

burst_hook_error

`dms::burst_hook_error``()`

Returns the last burst hook error message stored.

If called outside a hook, returns the value set by the last hook or an empty string if the last hook did not set an error message.

burst_hook_first_oid

dms::burst_hook_first_oid()

Returns the first OID in the current object. For graphic objects, returns the graphic element that contains the graphic.

Returns `oid_null()` if called outside a hook.

burst_hook_last_oid

dms::burst_hook_last_oid()

Returns the last OID in the current object.

Returns `oid_null()` if called outside a hook.

burst_hook_is_graphic

dms::burst_hook_is_graphic()

Specifies whether a function is called from a text or graphic hook.

Returns `1` if hook is being called for a graphic object. Returns `0` if called for a text object.

Returns `0` if called outside a hook.

burst_hook_sess

dms::burst_hook_sess ()

Returns the session handle, which can be used to call repository API functions.

Results are undefined if called outside a hook.

burst_hook_value

dms::burst_hook_value()

Returns the candidate object name or location as a string. The string is generated by internal rules, unless changed by the hook.

Returns an empty string if called outside a hook.

dobj_burst

dobj_burst (*dobj*)

Bursts the locked document object represented by the document object handle *dobj*. The bursting process follows the established defaults and document-type-definition-specific rules for the applicable document type. If the *dobj* contains sibling (that is, more than one) top-level elements, it will not be burst.

If *dobj* is invalid, or the burst fails, the function sets `$main::ioerr` with error information and returns a zero (0). If the burst is successful, the function returns a one (1).

dobj_checkin

dobj_checkin (*dobj*[, *ud*])

Checks in the locked document object possessing the document object handle of *dobj*. To properly update the revised document in the repository, you must save the object before calling `dobj_checkin`. All revisions to the document object are then written to the repository.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If *dobj* is invalid, or the check-in fails, the function sets `$main::ioerr` and returns a zero (`0`). If the check-in is successful, the function returns a document object handle. Note that the returned document object handle may not be the same document object handle that was passed to the function.

The `dobj_close` on page 1044 function must be called to close the document object after it has been processed. However, the returned document object may differ from the one passed to the function. The document object can be closed by implementing the following sample code.

```
local new_dobj = dobj_checkin(dobj);
if (dobj_valid(new_dobj))
   {
      # A new dobj was returned. Release the old one and
      # start using the new one.
      dobj_close(dobj);
   dobj = new_dobj;
}
```

dobj_class

dobj_class (*dobj*)

Returns the class for the document object possessing the ACL document object handle of *dobj*.

If *dobj* is invalid, or has an unknown class, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns in integer for the class. Possible class values are:

- `1` = Container (that is, document object with children)
- `2` = Document object (no children)
- `3` = Expanded file entity
- `4` = Unexpanded file entity
- `5` = File entity open for editing in a separate window
- `6` = Included object
- `7` = Fallback markup inserted for a failed XML inclusion

dobj_close

dobj_close (*dobj*)

Closes the document object possessing the ACL document object handle of *dobj*. If the document object has any children, the function closes them as well.

If *dobj* is invalid, or the close operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

dobj_collapse_oid

dobj_collapse_oid (*oid*, *collapse*, *allinst*)

This function either collapses (*collapse*=TRUE) or expands (*collapse*=FALSE) the parent document object that contains the object *oid*. If *allinst* is set to TRUE, the function operates on all instances of the parent document object.

If *oid* is invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

Note

Expanding explicitly forces a collapsed document object to be reloaded.

dobj_construct

dobj_construct (*logicalid.*,*doc*[, *reserved*[, *ud*]])

Creates an ACL document object handle for the existing document object indicated by *logicalid*. The dobj_close on page 1044 function must be called to close the *dobj* after it has been processed.

This function uses the following parameters:

- *logicalid* — unique identifier for an object
- *doc* — document used for context information; required even when you are constructing a document object handle for a folder
- *reserved* — for internal use only
- *ud* — can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

The following integers are returned:

- `0` — operation failed
- A `docobj` is returned if the function succeeds.

dobj_create

`dobj_create` (*`sesshdl`*`, `*`name`*`, `*`loc`*`, `*`opts`*`, `*`doc`*`[, `*`ver`*`[, `*`objtype`*`[, `*`ud`*`]]])`

Creates an empty document object in the repository. The `dobj_close` on page 1044 function must be called to close the document object after it has been processed.

The *sesshdl* and *name* parameters specify the ACL session handle and name the repository should use for the new object (respectively).

The *loc* parameter contains an adapter-specific pointer to the new object's parent object.

The *opts* parameter is a bitmask that sets any optional data for the new object. The bits in the *opts* parameter are:

- `0x001` = object is locked
- `0x002` = unused
- `0x004` = object is a folder
- `0x008` = object is container (document object with children)
- `0x010` = object consists of SGML content
- `0x020` = object consists of XML content
- `0x040` = reserved bit—do not use

- `0x080` = object consists of HTML content
- `0x100` = object consists of text content

The *doc* parameter contains the ACL document handle for the top-level document object.

The optional *ver* parameter specifies the version of the new document object.

The *objtype* parameter sets the adapter-specific object type for the new document object. These object types are defined in the documentation for your adapter.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns the ACL document object handle for the new document object.

dobj_create_subtree

`dobj_create_subtree` (*`sesshdl`*, *`name`*, *`loc`*, *`opts`*, *`firstoid`*, *`lastoid`*[, *`ver`*[, *`objtype`*[, *`ud`*]]])

Creates a subtree in the repository. The `dobj_close` on page 1044 function must be called to close the document object after it has been processed.

The *sesshdl* and *name* parameters specify the ACL session handle and name the repository should use for the new object (respectively).

The *loc* parameter contains an adapter-specific pointer to the new object's parent object.

The *opts* parameter is a bitmask that sets any optional data for the new object. The bits in the *opts* parameter are:

- `0x001` = object is locked
- `0x002` = unused
- `0x004` = object is a folder
- `0x008` = object is container (document object with children)
- `0x010` = object consists of SGML content
- `0x020` = object consists of XML content
- `0x040` = reserved bit—do not use
- `0x080` = object consists of HTML content
- `0x100` = object consists of text content

The *firstoid* and *lastoid* parameters define the range of the document to be included in the subtree.

The optional *ver* parameter specifies the version of the new document object.

The *objtype* parameter sets the adapter-specific object type for the new document object. These object types are defined in the documentation for your adapter.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns the ACL document object handle for the new subtree.

dobj_delete

dobj_delete (*dobj* [, *ud*])

Deletes the document object possessing the ACL document object handle of *dobj* . The object is marked as deleted in the repository.

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

dobj_encl_dobj

dobj_encl_dobj (*dobj*)

Returns the document object handle for the object that encloses the document object specified by *dobj*. If *dobj* is the top level object, the function returns a `0`. The `dobj_close` on page 1044 function must be called to close the document object after it has been processed.

dobj_first_dobj

dobj_first_dobj (*dobj*)

Returns the ACL document object handle for the first valid document object in the chain of document objects containing *dobj*.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0).

dobj_first_oid

dobj_first_oid (*dobj*)

Returns the first oid within the document object possessing the ACL document object handle of *dobj*.

If *dobj* is invalid, or the operation fails, the function returns `null_oid`.

dobj_get_attr

dobj_get_attr (*dobj*, *attr*[, *idx*])

Returns the value of metadata attribute *attr* for the object with the ACL document object handle of *dobj*. If the desired metadata attribute contains an array, use the optional *idx* to access the desired entry in the array.

To obtain a Repository API attribute, the *attr* parameter should have one of two formats:

- The ACL abstraction of the attribute. ACL attribute names have the format `IO_ATTR_`*NAME*, where *NAME* is different for a specific attribute. The attributes that can be used with this function are listed in the *ACL Attribute Name* column of the *Attributes names for developers* section of your adapter documentation. This function can accept `dobj` attribute names but will not accept `sess` attribute names.
- The repository's name for the attribute. If you supply an attribute name that does not have the `IO_ATTR_`*NAME* format, the function will pass the literal name through to the repository. Using this method, you can access attributes in the repository that are not abstracted in ACL.

If either the *dobj* or *attr* values are invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a NULL.

dobj_get_attrlist

dobj_get_attrlist (*dobj*, *array*)

Use this function to return metadata attribute values for the object with the ACL document object handle of *dobj*. This function scans the associative array named *array*, and for each index that matches the name of a native metadata attribute, fills that value in the array with the value for that attribute.

The indices used in *array* should have one of two formats:

- The ACL abstraction of the attribute. ACL attribute names have the format `IO_ATTR_`*NAME*, where *NAME* is different for a specific attribute. The attributes that can be used with this function are listed in the *ACL Attribute*

Name column of the *Attributes names for developers* section of your adapter documentation. This function can accept `dobj` attribute names but will not accept `sess` attribute names.

- The repository's name for the attribute. If you supply an attribute name that does not have the `IO_ATTR_`*`NAME`* format, the function will pass the literal name through to the repository. Using this method, you can access attributes in the repository that are not abstracted in ACL.

If the function executes successfully, it returns a one (1). If either the *dobj* or *array* values are invalid, or the operation fails, the function returns a zero (0).

Example

```
propvalue["IO_ATTR_IS_CONTAINER"] = "";
propvalue["IO_ATTR_LOCKED"] = "";
propvalue["IO_ATTR_USE_LATEST_REV"] = "";
ret = dobj_get_attrlist (dobj, "propvalue");
```

dobj_is_mod

dobj_is_mod (*dobj*)

Returns a boolean value indicating whether the document object *dobj* has been modified. If the *dobj* has been modified, the function returns a one (1). If *dobj* is invalid, or has not been modified, the function returns a zero (0).

dobj_is_my_lock

dobj_is_my_lock (*dobj*)

Returns a boolean value indicating whether the document object *dobj* is locked by the current user. If the *dobj* is locked by the current user, the function returns a one (1). If *dobj* is invalid, or is not locked by the current user, the function returns a zero (0).

dobj_last_oid

dobj_last_oid (*dobj*)

Returns the last oid within the document object possessing the ACL document object handle of *dobj*.

If *dobj* is invalid, or the operation fails, the function returns null_oid.

dobj_lock

dobj_lock (*dobj* , *lockflag*[, *ud*])

Locks, in the repository, the document object with the *dobj* ACL document object handle.

- *dobj* — The object handle of the document object to be locked. If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a the document handle of the locked object.
- *lockflag* — An integer that determines how the lock will take place. The available *lockflag* values are:
  - `1` — Destroys any existing lock in favor of the new lock (may not work with all document management systems). Use with extreme caution.
  - `2` — Locks the top level document.
- The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

The `dobj_close` on page 1044 function must be called to close the document object after it has been processed. However, the returned document object may differ from the one passed to the function. The document object can be closed by implementing the following sample code.

```
local new_dobj = dobj_lock(dobj, 2);
if (dobj_valid(new_dobj))
   {
      # A new dobj was returned. Release the old one and
      # start using the new one.
      dobj_close(dobj);
   dobj = new_dobj;
   }
```

dobj_logicalid

`dobj_logicalid` (*`dobj`*)

This function returns the logical identifier for the document object specified by the document object handle *dobj*.

If *dobj* is invalid, the function returns the string `<null>`. If the function is successful, it returns a logical identifier.

dobj_move

`dobj_move` (*`sourcedobj`* , *`destdobj`*[, *`ud`*])

This function moves the document object *sourcedobj* into the folder or cabinet object specified by *destdobj*.

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If either *sourcedobj* or *destdobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns the document handle of the object *sourcedobj*.

dobj_next_dobj

dobj_next_dobj (*dobj*)

This function closes the document object handle indicated by *dobj* and returns the document object handle for the next valid document object in the chain of document objects containing *dobj*.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0).

dobj_poid

dobj_poid (*dobj*)

Returns the persistent object ID (POID) for the document object handle of *dobj*.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns `<NULL>`.

dobj_save

dobj_save (*dobj*, *opts*[, *doc*[, *ud*]])

Saves a document object.

The *dobj* parameter specifies the ACL document object handle of the object to save. The *doc* parameter specifies the ACL document handle that contains the document object. If you do not specify a *doc*, an ACL document will be taken from the *dobj*, if possible.

The *opts* parameter is a bitmask that sets the option flags for the saving process. The bits in the *opts* parameter are:

- `0x001` = save attributes
- `0x002` = save content
- `0x004` = save declarations
- `0x008` = do not save any processing instructions

- `0x010` = save nothing, but reorder the entity reference list (even if it hasn't changed)
- `0x020` = save any modified children (including entities that were edited inline)

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If the operation fails, the function returns a zero (0). If the operation is successful, the function returns a one (1).

dobj_session

dobj_session (*dobj*)

Returns the session handle for the document object with the ACL document object handle of *dobj*.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns -1.

dobj_set_attr

dobj_set_attr (*dobj*, *attr*, *val*[, *idx*])

For the object with the ACL document object handle *dobj*, this function assigns the value *val* to the metadata attribute *attr*. If the desired metadata attribute contains an array, use the optional *idx* to set the desired entry in the array.

To set a Repository API attribute, the *attr* parameter should have one of two formats:

- The ACL abstraction of the attribute. ACL attribute names have the format `IO_ATTR_`*NAME*, where *NAME* is different for a specific attribute. The attributes that can be used with this function are listed in the *ACL Attribute Name* column of the *Attributes names for developers* section of your adapter documentation. This function can accept `dobj` attribute names but will not accept `sess` attribute names.
- The repository's name for the attribute. If you supply an attribute name that does not have the `IO_ATTR_`*NAME* format, the function will pass the literal name through to the repository. Using this method, you can access attributes in the repository that are not abstracted in ACL.

If either the *dobj* or *attr* values are invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

This function does not commit all attribute changes to the repository until the repository object is closed (using the user interface or the dobj_close on page 1044 function). To commit changes to the repository sooner, use the dobj_save on page 1051 function and specify the option to save attributes.

dobj_set_attrlist

dobj_set_attrlist (*dobj*, *array*)

Use this function to set metadata attribute values for the object with the ACL document object handle of *dobj*. This function scans the associative array named *array*, and for each index that matches the name of a native metadata attribute, the function sets the value for that attribute to the value in the *array*.

The indices used in *array* should have one of two formats:

- The ACL abstraction of the attribute. ACL attribute names have the format `IO_ATTR_`*NAME*, where *NAME* is different for a specific attribute. The attributes that can be used with this function are listed in the ACL Attribute Name column of the Attributes names for developers section of your adapter documentation. This function can accept `dobj` attribute names but will not accept `sess` attribute names.
- The repository's name for the attribute. If you supply an attribute name that does not have the `IO_ATTR_`*NAME* format, the function will pass the literal name through to the repository. Using this method, you can access attributes in the repository that are not abstracted in ACL.

If the function executes successfully, it returns a one (1). If either the *dobj* or *array* values are invalid, or the operation fails, the function returns a zero (0).

This function does not commit all attribute changes to the repository until the repository object is closed (using the user interface or the dobj_close on page 1044 function). To commit changes to the repository sooner, use the dobj_save on page 1051 function and specify the option to save attributes.

Example

```
attrlist["IO_ATTR_DTD_PUBLIC"] = \
 public_id(current_doc());
attrlist["IO_ATTR_NAME"] = name;
dobj_set_attrlist(dobj, "attrlist");
```

dobj_unlock

dobj_unlock (*dobj*[, *ud*])

Unlocks, in the repository, the object with the *dobj* ACL document object handle. Note that this function performs the unlock without performing a check-in.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If *dobj* is invalid, or the operation fails, the function sets `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns the document handle of the unlocked object.

The `dobj_close` on page 1044 function must be called to close the document object after it has been processed. However, the returned document object may differ from the one passed to the function. The document object can be closed by implementing the following sample code.

```
local new_dobj = dobj_unlock(dobj);
if (dobj_valid(new_dobj))
   {
      # A new dobj was returned. Release the old one and
      # start using the new one.
      dobj_close(dobj);
   dobj = new_dobj;
   }
```

dobj_valid

dobj_valid (*dobj*)

Validates the document object with the *dobj* ACL document object handle.

If *dobj* is invalid, or the operation fails, the function returns a zero (0). If the operation is successful, the function returns a one (1).

import_graphic_folder

import_graphic_folder(*srcFolder*[, *cmsFolder*[, *descend*]])

This function imports all of the graphics stored in a specified file system directory into the connected document repository.

The *srcFolder* parameter is the file system path to the directory containing the graphics to be imported. This parameter is required for batch session, but is optional for interactive sessions. If an interactive call does not provide a value for this parameter, a browse dialog box is invoked.

The optional *cmsFolder* parameter is the repository path to the location in which the imported graphic objects should be stored. If a no value is provided for this parameter, the following locations are searched and the first valid location is used:

- The current folder location in the repository **Browser**.
- The default location for new graphic objects in the system-wide burst configuration file
- The current user's repository home directory

The optional *descend* parameter indicates whether subdirectories in the *srcFolder* should be recursively processed. The default is `0`, which disables subdirectory processing. Set the value of this parameter to `1` to enable subdirectory processing.

The function returns `1` if all located graphic objects are successfully imported into the repository or `0` if an error occurs. Processing stops when an error occurs, but any graphic objects that were imported prior to the error are saved in the repository.

logicalid_to_poid

`logicalid_to_poid` (*`logical_id`*)

Returns the Persistent Object ID (POID) for the document object with the logical ID of *logical_id*.

If *logical_id* is invalid, or the operation fails, the function returns the supplied *logical_id* value .

oid_dobj

`oid_dobj` (*`oid`*`[,` *`obj`*`]`)

Returns the document object handle for the element specified by *oid* (if any). If *oid* is invalid, or the specified *oid* has no document object handle, the function returns a zero (`0`). If the operation is successful, the function returns the document object handle for the *oid*. The `dobj_close` on page 1044 function must be called to close the document object after it has been processed.

A document can have two types of objects: file entity references and other objects. *oids* representing file entity references may have 2 objects associated with them: one object and one file entity reference object.

If *obj* is `1`, then file entity objects will not be returned. If it is omitted or `0`, then a file entity object will be returned if there is one on the *oid*. This means that if the *oid* has both types of object attached, then the file entity object is returned if *obj* is omitted or `0`.

An XInclude *oid* can also have multiple objects attached. This method returns the innermost object on the *oid* if it is an XML inclusion. If you do not want the innermost object, use the `dobj_encl_dobj` function.

If *oid* is not a file entity reference, *obj* is ignored.

oid_encl_dobj

oid_encl_dobj (*oid*[, *obj*[, *pos*]])

Returns the document object handle that encloses the element specified by *oid* and *pos* (if any). If *oid* is invalid, or the specified *oid* is not within a document object, the function returns a zero (0). If the operation is successful, the function returns the document object handle enclosing the specified *oid*. The dobj_close on page 1044 function must be called to close the document object after it has been processed.

A document can have two types of objects: file entity references and other objects. An *oid* representing a file entity references may have 2 objects associated with it: one object and one file entity reference object.

If *obj* is 1, then file entity objects will not be returned. If it is omitted or 0, then a file entity object will be returned if there is one on the parent object *oid*. This means that if the parent *oid* has both types of object attached, then the file entity object is returned if *obj* is omitted or 0.

An XInclude *oid* can also have multiple objects attached. This method returns the innermost object on the parent object *oid* if it is an XML inclusion. If you do not want the innermost object, use the dobj_encl_dobj function.

If the parent object *oid* is not a file entity reference, *obj* is ignored.

pos can be any of the following values:

- 0 — (The default.) *pos* is immediately after the start tag identified by *oid*.
- -1 — *pos* is immediately before the end tag of the element identified by *oid*.
- -2 — *pos* is immediately before the start tag.
- -3 — If *oid* is a start tag, *pos* is immediately after the end tag. Otherwise, *pos* is at the end of the object.

oid_encl_poid

oid_encl_poid (*oid*[, *obj*[, *pos*]])

Returns the POID that encloses the element specified by *oid* and *pos* (if any). If *oid* is invalid, or the specified *oid* is not within a POID, the function returns a null string. If the operation is successful, the function returns the POID enclosing the specified *oid*.

A POID can have two document objects when the POID is a reference to a file entity. The POID will always have a document object handle for the contents of the file entity being referenced. If the reference POID itself is a top-level sibling in another object, it will also have a document object handle for that object.

If *obj* is `1`, then file entity objects will not be returned. If it is omitted or `0`, then a file entity object will be returned if there is one on the parent object *oid*. This means that if the parent *oid* has both types of object attached, then the file entity object is returned if *obj* is omitted or `0`.

pos can be any of the following values:

- `0` — (The default.) *pos* is immediately after the start tag identified by *oid*.
- `-1` — *pos* is immediately before the end tag of the element identified by *oid*.
- `-2` — *pos* is immediately before the start tag.
- `-3` — If *oid* is a start tag, *pos* is immediately after the end tag. Otherwise, *pos* is at the end of the object.

oid_poid

`oid_poid` (*`oid`*)(*`obj`*)

Returns the POID for the element specified by *oid* (if any). If *oid* is invalid, or the specified *oid* has no POID, the function returns a null string. If the operation is successful, the function returns the POID for the *oid*.

A POID can have two document objects when the POID is a reference to a file entity. The POID will always have a document object handle for the contents of the file entity being referenced. If the reference POID itself is a top-level sibling in another object, it will also have a document object handle for that object.

If *oid* is a file entity reference, omitting *obj* or setting *obj* to a value of `0` specifies that `oid_poid` will return the document object handle of the contents of the file entity. If *oid* is a file entity reference, setting *obj* to a value of `1` specifies that `oid_poid` will return the document object handle of the reference POID.

If *oid* is not a file entity reference, *obj* is ignored.

oid_set_dobj

`oid_set_dobj` (*`oid`*, *`dobj`*)

Associates the element *oid* with the document object *dobj*.

If *oid* or *dobj* are invalid, the function writes error data to `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

path_session

`path_session` (*`path`*)

If the specified *path* is the correct logical id format for a connected repository, `path_session` returns the Arbortext Editor session handle for communicating with that repository. If the format of the specified *path* is not correct for any connected repositories, `path_session` returns zero (0).

poid_exists

`poid_exists` (*`poid`*)

Verifies the existence of the document object with a given *poid* within any of the currently active repositories.

If *poid* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

poid_first_oids

`poid_first_oids` (*`poid, arr`*)

Returns a non-associative array *arr* containing a list of the OIDs for the first siblings to the specified *poid*.

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0).

poid_list_children

`poid_list_children` (*`logicalid, child_array`*`[, ud ]`)

Builds an array *child_array* containing data for the children (if any) of the document object *logicalid*.

The ACL array *child_array* will contain the following data.

- Type of POID
- Name of document object
- Revision number of the POID
- Is the object locked (Locked = True, Unlocked = False)?

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *logicalid* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns `-1`. If the operation is successful, the function returns the number of entries added to the output array.

poid_list_parents

poid_list_parents (*logicalid*, *parent_array*[, *ud*])

Builds an array *parent_array* containing the POIDs that contain the document object *logicalid*.

The ACL array *parent_array* will contain the following data.

- Type of POID (usually `C` for container).
- Name of document object.
- POID of the document object.
- Revision number of the POID

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *logicalid* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns `-1`. If the operation is successful, the function returns the number of entries added to the output array.

poid_list_revs

poid_list_revs (*logicalid*, *rev_array*[, *ud*])

Builds an array *rev_array* containing revision data for the document object *logicalid*.

The ACL array *rev_array* will contain the following data.

- Type of POID
- Name of document object
- Revision number of the POID
- Is the object locked (Locked = True, Unlocked = False)?

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *logicalid* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns `-1`. If the operation is successful, the function returns the number of entries added to the output array.

poid_list_search_attr

poid_list_search_attr(*sesshdl*, *query*, *result_array*[, *ud*])

Builds an array *result_array* containing data for all document objects (if any) within the repository specified by *sesshdl*, using the query *query*, and matching the object info within *result_array*.

The ACL array *result_array* can contain the following data.

- Type of POID
- Name of document object
- Revision number of the POID
- Is the object locked (Locked = True, Unlocked = False)?

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If the operation fails, the function writes error data to `$main::ioerr` and returns `-1`. If the operation is successful, the function returns the number of entries added to the output array.

poid_list_search_text

`poid_list_search_text``(`*`sesshdl`*`, `*`query`*`, `*`result_array`*`[, `*`ud`*`])`

Builds an array *result_array* containing data for all document objects (if any) within the repository specified by *sesshdl*, using the query *query*, and matching the object info within *result_array*.

The ACL array *result_array* can contain the following data.

- Type of POID
- Name of document object
- Revision number of the POID
- Is the object locked (Locked = True, Unlocked = False)?

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If the operation fails, the function writes error data to `$main::ioerr` and returns `-1`. If the operation is successful, the function returns the number of entries added to the output array.

poid_to_logicalid

`poid_to_logicalid` `(`*`poid`*`[, `*`verlabel`*`])`

This function accepts the persistent object identifier (POID) *poid* and the optional version label *verlabel* and returns a logical identifier. The logical identifier is used to represent a particular binding or version of a document object. The function returns the logical identifier on success and an error code (that is, less than zero) on error.

sess_add_callback

sess_add_callback (*sesshdl*, *fcode*, *fname*)

For the currently active session *sesshdl*, this function sets a callback for the session function *fcode* to call the ACL function *fname*.

The legal session *fcode*s are:

- autosave on page 1076
- checkin on page 1076
- construct on page 1076
- create on page 1077
- getfile on page 1077
- lock on page 1078
- postburst on page 1079
- preburst on page 1078
- precheckin on page 1079
- putfile on page 1079
- save on page 1080
- unlock on page 1080

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

sess_bursting

sess_bursting (*sess*)

This function assesses whether the session identified by *sess*, the currently active session handle, is currently bursting a document into the repository.

The function returns 1 if true, 0 if not true, or a negative value of the session handle is invalid.

The function is intended to be used in a callback or hook function which might be called when bursting is active since it will return 0 otherwise.

sess_clear_burst_config

sess_clear_burst_config(*session*)

Clears out all burst configuration settings that have been loaded for the given repository *session* and reloads the system-wide burst configuration settings. Document-type-specific burst configurations are then loaded as needed, for example, when a document of that document type is burst.

You can use this function while developing your burst configuration files. The function enables you to test new burst configuration settings without having to exit Arbortext Editor. The function does not change the location that Arbortext Editor uses to load burst configuration files.

Note

Exercise caution when using this function, as errors in a burst configuration file will cause problems for future bursting.

sess_connect

sess_connect (*dlid*, *id*, *pw*, *repid*[, *ud*])

Note

This function performs one of the many steps necessary to connect a user to a repository. If you are writing a script that connects to a repository, you should use the sess_start on page 1072 function, as it will perform all the connection steps in their proper order.

Connects the user *id* using a password of *pw* to the repository *repid* with the dynamic library ID of *dlid*.

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If the operation fails, the function places error data in `$main::ioerr` and returns -1. If successful, the function returns a new session handle for the connection.

sess_connected

sess_connected (*adapterName*, *dmsid*)

Checks whether the session is connected to the specified adapter and dmsid. `sess_connected` returns `1` if a session exists. Otherwise, it returns `0`.

The following table summarizes the parameter values.

Parameters for `sess_connected` function

Parameter	Description
adapterName	String identifier for adapter (case sensitive).
dmsid	Optional repository identifier. If not supplied, any session for *adapterName* is matched.

sess_disconnect

`sess_disconnect` (*`sesshdl`*)

Disconnects the session *sesshdl*.

If *sesshdl* is invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

> **Note**
>
> This function performs one of the many steps necessary to disconnect a user from a repository. If you are writing a script that disconnects from a repository, use the sess_end on page 1065 function, as it will perform all the disconnection steps in their proper order.

sess_doc_burst

`sess_doc_burst` (*`sesshdl`*[, *`doc`*[, *`flags`*[, *`name`*[, *`folder`*[, *`fid`*]]]]])

Bursts the specified document into an established repository session. The document must be a file system object. The *sesshdl* parameter is the currently active session handle. The *doc* specifies the document to burst. If zero (0), the current document is used. The *flags* parameter specifies any desired overrides to default bursting options and is published by ORing the following bits:

- `0x0001` — Enable full text indexing on the top most object.
- `0x0002` — Import file entities.
- `0x0004` — Do not import file entities.

- `0x0008` — Import graphic files.
- `0x0010` — Do not import graphic files.
- `0x0020` — Burst on element boundaries.
- `0x0040` — Ignore element boundaries.
- `0x0080` — Use the file name for the topmost object name.
- `0x0100` — Do not use the file name for the topmost object name.
- `0x0400` — Lock the topmost object for editing.
- `0x0800` — Do not lock the topmost object.
- `0x1000` — Use location rules for child objects even when `useroverride="on"` in the burst configuration file.

The negative flag settings override session defaults.

The *name* parameter gives a user-specified name to use for the topmost object. If the value is a null string, then the bursting rules are consulted.

The *folder* parameter gives a user-specified destination folder for all objects created. If your system-wide bursting configuration specification has the `useroverride` set to `on`, then the folder parameter gives a user-specified destination folder for all objects created. If a null string, then the bursting rules determine in which folders the new objects are created. If your system-wide bursting configuration specification has the `useroverride` set to `off`, then the folder parameter is not used and the bursting rules determine in which folders the new objects are created.

The *fileid* is an ACL file handle as returned by `open()` to use for logging information and error messages. If *fileid* is greater than or equal to zero (`0`), then no logging is performed.

If the *dobj* contains sibling (that is, more than one) top-level elements, it will not be burst. If *dobj* is invalid, or the burst fails, the function sets `$main::ioerr` with error information and returns a zero (`0`). If the burst is successful, the function returns a one (`1`).

sess_element_is_boundary

`sess_element_is_boundary``(`*`sesshdl`*`,` *`elementname`*`[,` *`doc`*`])`

For the repository session *sesshdl*, this function inspects the specified *elementname* for the specified *doc* and returns a value indicating the following:

- Whether or not the *elementname* is a bursting boundary.
- If the *elementname* is a bursting boundary, should it be burst as a file entity reference or a virtual document child.

The *sesshdl* parameter is a session handle. The *elementname* is the name of an element within the document's document type definition (DTD). The optional parameter *doc* is a document identifier for the desired document. If omitted, the return value for `current_doc` is used.

The possible return values are:

- `0` — *oid* is not a bursting boundary.
- `1` — *oid* is a bursting boundary and should be a file entity reference.
- `2` — *oid* is a bursting boundary and should be a virtual document child.
- `3` — *oid* is a bursting boundary and should be an XML inclusion reference.

sess_end

`sess_end` (*`hSess`*)

Disconnects a user from a repository. This function is the preferred method of disconnecting users from repositories using ACL. The *hSess* parameter is the session handle returned by `sess_start`. `sess_end` returns `1` if the session is successfully ended. If an error occurs during disconnection, such as encountering an invalid handle, `sess_end` returns `0`.

sess_err_message

`sess_err_message` (*`sesshdl`*)

Returns the session error message. Use this function to obtain the session-specific error message when another Repository API function returns an error code of `-183`. The *sesshdl* parameter is a session handle.

sess_extension

`sess_extension` (*`sesshdl`*, *`opcode`*, *`argArray`*)

This function invokes a repository adapter-specific extension function. Some adapters provide functionality beyond the standard ACL repository API. Each function is identified by an integer *opcode* that is defined by the adapter. Extension functions take an array of strings as arguments and return a string. `sess_extension` returns the result of the *opcode* extension method, which is typically a string or number.

- *sesshdl* is a session handle.
- *opcode* is an integer that identifies which extension method to invoke.
- *argArray* is an array of argument values. The number of parameters and their meaning depends on the specific extension function.

If *opcode* is invalid or the extension method fails, `sess_extension` writes error data to *$main::ioerr*. The value returned on failure depends on the specific extension method.

Note

This function is not supported by the PTC Server connection. You must use the Arbortext Object Model (AOM) **CMSSession.invokeExtension** method instead. Refer to the *Programmer's Reference* for more information about the AOM interfaces.

sess_get_attr

sess_get_attr (*sesshdl*, *attr*)

Returns the value for the session attribute *attr* within the currently active session *sesshdl*.

To obtain an attribute value, *attr* should have a format of `IO_ATTR_`*NAME*, where *NAME* is different for a specific attribute. The session attributes that can be used with this function are listed in the `ACL Attribute Name` column of the *Attributes names for developers* section of your adapter documentation. This function can accept `sess` attribute names and will not accept `dobj` attribute names.

If the specified *attr* is a repeating attribute, you can supply a string with the attribute name and array index (for example, *attrname*`[`*idx*`]`) to access the desired entry in the array.

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0).

sess_get_create_info

sess_get_create_info (*sesshdl*, *return_array*)

For the session *sesshdl*, this function builds an array *return_array* containing the system-wide object creation information that is not specific to a document type definition (DTD). The *return_array* is an associative array and can be used to build a user interface that shows default values for a new object and allows the user to override those defaults.

The array is the same for all adapters. Some adapters do not use all the indices in the array; these indices will be left empty.

The ACL array *return_array* will contain the following data:

Array index	Value description
IO_CRE_FILE_REFERENCE	Determines whether **Insert** and **Share Object** will create an entity or an XML inclusion. Possible values are `xinclude` or `entity`.
IO_CRE_FULL_TEXT_SEARCH	Determines whether the object is flagged for full text searching.
IO_CRE_LATEST_ID	Default version label that indicates that an object is the official current version.
IO_CRE_LOGICAL_ID	Default version label that tells the adapter to load the working copy of an object, followed by the "official" current copy (if no working copy exists).
IO_CRE_MAX_LEN	Default maximum name length for document objects.
IO_CRE_ROOT_TYPE	Root object type for Arbortext Editor objects.
IO_CRE_TEMP_ID	Default version label that indicates that an object is a working copy.
IO_CRE_TOP_LOCKED	Determines whether the topmost document object is locked for editing after bursting.

If *sesshdl* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

sess_get_graphic_create_info

`sess_get_graphic_create_info``(`*`sesshdl`*`, `*`asisstart`*`, `*`return_array`*`)`

For the session *sesshdl*, this function builds an array *return_array* containing the default creation information for a new graphic object. The *return_array* is an associative array and can be used to build a user interface that shows default values for a new graphic object and allows the user to override those defaults.

The *asisstart* parameter is the starting node for the new graphics object.

The ACL array *return_array* will contain the following data:

Array index	Value description
IO_CRE_LOCATION	Default location for new graphics.
IO_CRE_OBJECT_TYPE	Default object type for graphics.
IO_CRE_LABEL	Default version label for graphics.

If *sesshdl* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

sess_get_obj_create_info

sess_get_obj_create_info(*sesshdl*, *asisstart*, *asisend*, *top*, *return_array*)

For the session *sesshdl*, this function builds an array *return_array* containing the default creation information for a new object. The *return_array* is an associative array and can be used to build a user interface that shows default values for a new object and allows the user to override those defaults.

The *asisstart* and *asisend* parameters define the starting and ending nodes for the new object.

The *top* parameter indicates whether the “topmost is filename” naming rule is being used.

The array is the same for all adapters. Some adapters do not use all the indices in the array; these indices will be left empty.

The ACL array *return_array* will contain the following data:

Array index	Value description
IO_CRE_NAME	Default name for the new object, according to the default naming rules.
IO_CRE_LOCATION	Default location for the new object.
IO_CRE_OBJECT_TYPE	Default object type for the new object.
IO_CRE_LABEL	Default version label for the new object.

If *sesshdl* is invalid, or the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If the operation is successful, the function returns a one (1).

sess_get_file

sess_get_file (*poid*, *opts*[, *rendition* [, *ud*]])

Returns the name of a local file that contains the specified *poid* using the read option *opts*. An *opts* entry of one (1) reads the file as “read-only”; an *opts* entry of two (2) reads the file as “read\write”.

Use the optional *rendition* parameter to specify the desired rendition when retrieving the contents of a graphic object. The *rendition* must match one of the Notations declared in your DTD.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0).

sess_get_file_poid

`sess_get_file_poid` (*`sesshdl`*, *`filename`*)

This function is used to query the **entity lookup table**. During document bursting, Arbortext Editor refrains from adding the same file-based entity to the document repository more than once, preferring to use the exiting object in the repository. The **entity lookup table** maintains a list of the file entities already added and their respective persistent object identifiers (POIDs) in the repository.

The *sesshdl* parameter is a session handle. The *filename* parameter is the path to the file entity which has been added to the repository.

This function returns the POID of the document object matching the provided *filename* and a zero (0) if the *filename* was not in the table.

sess_info

`sess_info` (*`sesshdl`*)

Returns information for the session *sesshdl*, including its POID prefix. The resulting string of returned data will consist of several fields, separated by “;” delimiters.

If *sesshdl* is invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0).

sess_initialize

`sess_initialize` (*`library`*[, *`ud`*])

Note

This function performs one of the many steps necessary to connect a user to a repository. If you are writing a script that connects to a repository, you should use the sess_start on page 1072 function, for it will perform all the connection steps in their proper order.

Sets up a session that could connect to a repository. The function uses adapter data from the library located in the file name (including path) specified by *library*.

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *library* is invalid, or the operation fails, the function places error data in `$main::ioerr` and returns -1. If successful, the function returns a *dllid* (dynamic loading library identification) for the library.

sess_is_boundary

`$ret=`**`sess_is_boundary`**`(`*`sesshdl`*`, `*`oid`*`)`

For the repository session *sesshdl*, this function inspects the specified *oid* and returns a value indicating the following:

- Whether or not the *oid* is a bursting boundary.
- If the *oid* is a bursting boundary, should it be burst as a file entity reference or a virtual document child.

The *sesshdl* parameter is a session handle. The *oid* parameter is an object identifier for the specific object location in the document.

The possible return values are:

- `0` — *oid* is not a bursting boundary.
- `1` — *oid* is a bursting boundary and should be a file entity reference.
- `2` — *oid* is a bursting boundary and should be a virtual document child.
- `3` — *oid* is a bursting boundary and should be an XML inclusion reference.

sess_put_file

`$poid = `**`sess_put_file`**`(`*`sesshdl`*`, `*`filename`*`, `*`objectname`*`, `*`folder`*`, `*`rendition`*`[, `*`ud`*`])`

This function creates a new object in the document repository from the contents of *filename*. The function is intended to be used to add graphic files to the repository. To move an SGML, XML or HTML document into the repository, use the document bursting function (for example, dobj_burst on page 1042).

The *sesshdl* parameter is a session handle. The *objectname* parameter specifies the name for the new object in the repository. Use the *folder* parameter to specify a destination location (in the repository) for the new object. Use the *rendition* parameter to specify the type of data that is being saved. The *rendition* must match one of the Notations declared in your DTD.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. Refer to Repository API Callback Functions on page 1074 for more information on using callbacks with the Repository API functions.

The function returns a logical ID on sucess.

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0).

sess_set_attr

`sess_set_attr` (*`sesshdl`*, *`attr`*, *`value`*)

Within the currently active session *sesshdl*, this function sets the attribute *attr* to the value *value*.

If the specified *attr* is a repeating attribute, you can supply a string with the attribute name and array index (for example, *`attrname[idx]`*) to set the desired entry in the array.

To set an attribute value, *attr* should have a format of `IO_ATTR_`*`NAME`*, where *NAME* is different for a specific attribute. The session attributes that can be used with this function are listed in the `ACL Attribute Name` column of the *Attributes names for developers* section of your adapter documentation. This function can accept `sess` attribute types and will not accept `dobj` attribute types.

If the operation fails, the function writes error data to `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

sess_set_file_poid

`sess_set_file_poid` (*`sesshdl`*, *`filename`*, *`poid`*)

This function is used to add an entry to the **entity lookup table**. During document bursting, Arbortext Editor refrains from adding the same file-based entity to the document repository more than once, preferring to use the exiting object in the repository. The **entity lookup table** maintains a list of the file entities already added and their respective persistent object identifiers (POIDs) in the repository.

The *sesshdl* parameter is a session handle. The *filename* parameter is the path to the file entity which has been added to the repository. The *poid* parameter is the POID of the file entity in the repository.

This function returns one (`1`) on success and zero (`0`) on failure. If the file is already in the table, it returns a zero (`0`) and displays the following message: `Table entry found for` *`filename`* `(`*`poid`*`), no new table entry added.`

sess_share

`sess_share` `(`*`sesshdl`*`,` *`OID_start`*`,` *`OID_end`*`,` *`folder`*`[,` *`ud`*`])`

Shares a range of elements (from *OID_start* to *OID_end*) as an object within the indicated session *sesshdl*. If necessary, the adapter will burst the selected range of elements according to the current burst configuration. The *folder* parameter is a string representing the path and name of the parent directory for the document to be created. `sess_share` uses the following bursting rules to determine whether an XML inclusion or entity reference should be used when creating a new object:

1. If the document type-specific burst configuration contains an object boundary rule for the element with an `objecttype` of `entity` or `xinclude`, that value is used.
2. If the first rule doesn't apply, and the `createdefaults` element in the system-wide burst configuration specifies a `filereference` attribute, that value is used.
3. If neither of these rules applies, then the value of the `-filereference` set option is used.

   Included objects will always be parsed (the parsed attribute is not set, so it defaults to `xml`).

The *ud* parameter passes user defined data through to the adapter-specific implementation of this function.

If *sesshdl*, *OID_start*, or *OID_end* are invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0). If successful, the function returns the ACL document object handle for the new document object.

The `dobj_close` on page 1044 function must be called to close the document object after it has been processed.

sess_start

`sess_start` `(`*`adapt`*`,` *`userid`*`,` *`pw`*`,` *`dmsid`*`,` *`adptdata`*`,` *`mode`*`,` *`win`*`)`

Connects a user to a repository. This function is a wrapper for other functions such as modify_graphic_entities on page 682 and sess_initialize on page 1069 and is the preferred method of connecting users to repositories using ACL.

The following table summarizes the parameter values.

Parameters for `sess_start` function

Parameter	Description
adapt	String identifier for adapter (case sensitive). For the PTC Server connection: `PTC Server`
userid	User ID for desired repository.
pw	User's password for access to desired repository.
dmsid	Identifier for repository. For the PTC Server connection: URL to the PTC Server.
adptdata	Adapter data. For the PTC Server connection: Use the context and workspace separated by a comma. For example: `"context= ProductA,workspace=ws1"`
mode	Set to one of two values: `interactive` or `batch`. Use the `interactive` setting if the function is being called as a result of a user interface event (for example, clicking a connect button on a dialog). Use the `batch` setting if you are connecting to the repository using a batch script. If no value (that is, `""`) is given, *mode* will default to `batch`.
win	If the *mode* parameter is set to `interactive`, set the *win* parameter to the window handle for the dialog that called `sess_start`. If the *mode* parameter was set to anything other than `interactive`, set this value to zero (`0`).

If the operation fails, the function places error data in `$main::ioerr` and returns -1. If successful, the function returns a new session handle for the connection.

When disconnecting a user from a repository, use the sess_end on page 1065 function to perform all the disconnection steps in their proper order.

sess_terminate

`sess_terminate` (*`dllid`*)

Unloads the repository adapter specified by *dllid*.

If *dllid* is invalid, or the operation fails, the function places error data in `$main::ioerr` and returns a zero (0). If successful, the function returns a one (1).

sess_user_override

sess_user_override (*sesshdl*)

This function returns a one (1) if the session specified by *sesshdl* has the user override set on for bursting-related options such as object names. The function returns a zero if there is an error or if the specified session does not have user override set. The *sesshdl* parameter is a session handle.

set_burst_hook_error

dms::set_burst_hook_error(*msg*)

Called by hooks to set the error message to *msg* and stop the burst. The hook will continue to execute until it returns.

Has no effect if called outside a hook.

set_burst_hook_value

dms::set_burst_hook_value(*value*)

Called by a hook to set the new candidate object name or location to *value*.

Has no effect if called outside a hook.

Repository API Callback Functions

Adapter integrators may find it necessary to expand the functionality of the default Repository Adapter behavior by using ACL functions that Arbortext Editor executes before calling the equivalent function within the Repository API. The following table lists the callback functions and their corresponding Repository API function(s):

Callback function	Repository API function
autosave	N/A
create	dobj_create and dobj_create_subtree
checkin	dobj_checkin
construct	dobj_construct
getfile	sess_get_file

Callback function	Repository API function
lock	dobj_lock
postburst	sess_doc_burst and dobj_burst
preburst	sess_doc_burst and dobj_burst
precheckin	dobj_checkin
putfile	sess_put_file
save	dobj_save
unlock	dobj_unlock

The ACL startup file for the adapter registers callbacks for events such as autosave, checkin, lock, and getfile. This allows the adapter to, for example, save an object if the autosave preference has been set, extend checkin to capture user comments before the actual check-in occurs, instruct lock to inquire about version selection and allow getfile to accommodate different graphic rendition formats.

The *ud* parameter can be used to pass user defined information to a callback; strings must match the adapter encoding. The Adapters use 16–bit Unicode strings.

Registering Callbacks

Use sess_add_callback on page 1061 to register the ACL callback for a particular Repository API event. The following examples demonstrate how to register callback events to functions from the sample package rep_api.

```
# Override the autosave, checkin, lock and getfile behavior.
 sess_add_callback(sess, "autosave", "rep_api::autosave");
 sess_add_callback(sess, "checkin", "rep_api::checkin");
 sess_add_callback(sess, "lock", "rep_api::checkout");
 sess_add_callback(sess, "getfile", "rep_api::getfile");
```

Note

Callbacks are typically registered in the section of code that starts the session. For example, if an adapter-specific connect() function has been defined, it will be called just after the session is established.

Refer to Repository API Callback Functions on page 1074 for a list of callback types and their related repository API functions.

autosave

autosave (*docid*)

The ACL function registered for this event will be called if the autosave preference has been selected in Arbortext Editor. The autosave callback must be provided in order for `autosave` to include repository objects.

- *docid* — the document id

The `autosave` callback returns the following integer values:

- `0` — the object has not been autosaved
- anything other than `0` — the object has been autosaved

checkin

checkin(*dobj*, *docid*, *ud*)

This callback corresponds to the Repository API function dobj_checkin on page 1043 and checks in the specified document object. This callback is called after the *dobj* has been burst. The precheckin on page 1079 callback is called before the *dobj* has been burst.

- *dobj* — an ACL document object handle for the document object to check in
- *docid* — the document used for context information
- *ud* — user defined information; strings must match the adapter encoding

The `checkin` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the normal check-in can be performed
- greater than `0` — the callback succeeded and has returned the *docobj*

construct

construct (*poid*, *docid*, *reserved*, *ud*)

This callback corresponds to the dobj_construct on page 1044 function and constructs a document object handle (`docobj`) for the document object in the repository.

- *poid* — the logical ID for the repository object
- *docid* — the document used for context information; required even when you are constructing a document object handle for a folder
- *reserved* — for internal use only
- *ud* — user defined information; strings must match the adapter encoding

The `construct` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the normal construct routine should be performed
- greater than `0` — the callback succeeded and has returned the *docobj*

create

`create``(`*`name`*`, `*`folder`*`, `*`opts`*`, `*`first`*`, `*`last`*`, `*`version`*`, `*`objtype`*`, `*`ud`*`)`

This callback corresponds to the `dobj_create` on page 1045 and `dobj_create_subtree` on page 1046 functions and creates a new document object.

- *name* — a string representing the name of the document being created.
- *folder* — a string representing the path of the parent folder for the new document
- *opts* — a complete list of the available bitmask options is available in the `dobj_create` help topic
- *first* — the object ID (oid) of the first object in the document
- *last* — the object ID (oid) of the last object in the document
- *ver* — the version of the new document object
- *objtype* — the repository-specific object type for the new document object as defined in your adapter documentation
- *ud* — user defined information; strings must match the adapter encoding

The `create` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the normal create routine should be performed
- greater than `0` — the callback succeeded and has returned the *docobj*

getfile

`getfile``(`*`poid`*`, `*`opts`*`, `*`rendition`*`, `*`ud`*`)`

This callback corresponds to the sess_get_file on page 1068 function and retrieves an object (usually a graphic object) from the document repository. This callback can be used to override the rendition of the file that is retrieved.

- *poid* — the logical ID for the repository object
- *opts* — an integer value representing the desired operation

The following *opts* values are available:

  - `1` indicates a request to read the contents
  - `2` indicates a request to write to the contents

- *rendition* — the desired rendition of a graphic object; value must match one of the notations declared in your data model
- *ud* — user defined information; strings must match the adapter encoding

The `getfile` callback returns the following string:

- `"IO_ERR_NOMORE"` — the normal `getfile` routine should be performed
- If any other string is returned, the callback succeeded and the returned string represents the name of the local file that was created. The normal `getfile` function should not be performed.

lock

`lock``(`*`dobj`*`, `*`lockflags`*`, `*`docid`*`, `*`ud`*`)`

This callback corresponds to the dobj_lock on page 1049 function and locks the specified document object.

- *dobj* — an ACL document object handle for the document object to lock
- *lockflags* — a list of options for this parameter are available in the dobj_lock on page 1049 function help topic
- *docid* — the document used for context information
- *ud* — user defined information; strings must match the adapter encoding

The `lock` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the normal lock routine should be performed
- greater than `0` — the callback succeeded and has returned the *docobj*

preburst

`preburst` `(`*`doc`*`)`

This callback is invoked immediately before a document is burst. Bursting may occur when a document is imported, saved, or checked in, depending on the repository adapter's bursting policy. The `preburst()` callback may not perform the bursting itself. The *doc* parameter specifies the document identifier for the document to burst.

`preburst()` returns the following integer values:

- `0` — The burst can proceed.
- `< 0` — An error has occurred.

precheckin

precheckin (*dobj*, *docid*, *ud*)

This callback corresponds to the dobj_checkin on page 1043 function and is called just before the *dobj* is burst. The checkin on page 1076 callback is called after the *dobj* is burst.

- *dobj* — an ACL document object handle for the document object to check in
- *docid* — the document used for context information
- *ud* — user defined information; strings must match the adapter encoding

The `precheckin` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the check-in can proceed
- greater than `0` — the callback succeeded and has returned the *docobj*

postburst

postburst (*succeeded*, *doc*)

This callback is invoked immediately after a document is burst. It is always called, regardless of whether the burst succeeded. Bursting may occur when a document is imported, saved, or checked in, depending on the repository adapter's bursting policy.

Errors reported by post-burst hooks are not fatal. If errors are reported, the burst is still considered successful and `sess_doc_burst()` will not return an error. *succeeded* has the following values:

- `1` — The burst succeeded.
- `0` — A bursting error occurred.

`postburst(succeeded)` returns the following integer values:

- `0` — The burst can proceed.
- `< 0` — An error has occurred.

The *doc* parameter specifies the document identifier for the document to burst.

putfile

putfile(*filename*, *objectname*, *folder*, *rendition*, *ud*)

This callback corresponds to the sess_put_file on page 1070 function and adds a file (usually a graphic file) to the document repository.

- *filename* — the name of the file to add to the repository
- *objectname* — the name of the new object to add to the repository
- *folder* — the location for the new object in the repository
- *rendition* — the desired rendition of a graphic object; value must match one of the notations declared in your data model
- *ud* — user defined information; strings must match the adapter encoding

The putfile callback returns the following string:

- "IO_ERR_NOMORE" — the normal putfile routine should be performed
- If any other string is returned, the callback succeeded and the returned string represents the logical ID of the object that was created. The normal putfile function should not be performed.

save

save(*dobj*, *first*, *last*, *opts*, *ud*)

This callback corresponds to the dobj_save on page 1051 function and saves a document.

- *dobj* — an ACL document object handle for the document object to save
- *first* — the object ID (oid) of the first object in the document
- *last* — the object ID (oid) of the last object in the document
- *opts* — a complete list of the available bitmask options is available in the dobj_save on page 1051 help topic
- *ud* — user defined information; strings must match the adapter encoding

The save callback returns the following integer values:

- less than 0 — an error has occurred
- 0 — the normal save should proceed
- greater than 0 — the callback succeeded and has returned the *docobj*

unlock

unlock(*dobj*, *docid*, *ud*)

This callback corresponds to the dobj_unlock on page 1053 function and unlocks the desired document object.

- *dobj* — an ACL document object handle for the document object to lock
- *docid* — the document used for context information
- *ud* — user defined information; strings must match the adapter encoding

The `unlock` callback returns the following integer values:

- less than `0` — an error has occurred
- `0` — the normal unlock should proceed
- greater than `0` — the callback succeeded and has returned the *docobj*

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z