

# 21

## Custom MUI Dialogs

Copyright 1996-1999. Bruce A. Epstein. All rights reserved.

This bonus chapter covers creating custom dialog boxes with the MUI Dialog Xtra. The latest version of this document can be found in Acrobat PDF format at <http://www.zeusprod.com/nutshell/chapters/muixtra.html>. This chapter is a superset of Chapter 15, *The MUI Dialog Xtra*, in *Lingo in a Nutshell*.

The MUI Xtra was introduced in Director 6 and can create a variety of custom dialog boxes, including Alert boxes. It is used during authoring by Director and other Xtras to generate custom dialog boxes. It is installed in the *Xtras/Media Support* subfolder by default.

If using any of the MUI Xtra's methods to create custom dialog boxes at runtime, you must include the Xtra named "MUI Dialog" (for Macintosh) or "MUI Dialog.x32" (for Windows) with your Projector, presumably in an *Xtras* folder one level down from your Projector. (I recommend against bundling Xtras into Projectors. See Chapter 10, *Using Xtras*, in *Director in a Nutshell* for details.)

The MUI Xtra's main limitation is that it does not support Windows 3.1 (for users of D6.5 and prior versions), nor is it available for Shockwave. If you are using 16-bit (Windows 3.1) Projectors, consider using the `baMsgBox` function in Buddy API (<http://www.mods.com.au/budapi/>), which can create simple dialogs. The MUI Xtra is not marked "Safe for Shockwave", so Shockwave developers should use one of the alternatives discussed in the Chapter 14, *Graphical User Interface Components*, in *Director in a Nutshell*.

Obtain the latest version of the MUI Dialog Xtra for your version of Director as part of the Director 6.0.2, Director 6.5, or Director 7.0.2 update from <http://www.macromedia.com/support/director/updown/updates.html>.

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

---

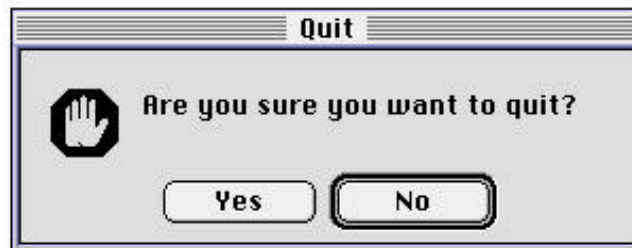
This chapter was tested using MUI Dialog Xtra version 6.0.2r33 dated December 12, 1997. Only minor differences, and perhaps some bug fixes, are expected in later versions.

---

The D6.0.2 updater may fail if you had installed any of the Macromedia Xtras that installed a later version of the MUI Xtra. Update to D6.0.2 before installing Macromedia Xtras such as the Flash Asset Xtra or Custom Cursor Xtra, which were sold separately for a brief time before D6.5 was released, and which included interim releases of the MUI Xtra.

## Simple MUI Alert Dialogs

The MUI Xtra's *Alert()* command creates modal dialogs with up to three buttons. As with the standard Lingo *alert* command, it's text is limited to 255 characters, so you'll need to create a custom dialog (see below) for longer text. *Alert()* accepts a property list defining the button choices, the text to display, etc. Example 21-1 creates the dialog box shown in Figure 21-1.



*Figure 21-1: Two-Button Alert Dialog*

*Example 21-1: Standard Multi-button MUI Alert Dialog*

```
on testMUIalert
  -- Create an instance of the MUI Xtra
  set MUIobject = new (xtra "MUI")
  -- The MUI Alert doesn't beep, so let's do so manually.
  beep
  -- Define the attributes of the dialog in a property list
  set alertPropertiesList = ⌗
    [#buttons: #YesNo, ⌗
     #default: 2, ⌗
     #title: "Quit", ⌗
     #message: "Are you sure you want to quit?", ⌗
     #movable: TRUE, ⌗
     #icon: #stop]
  -- Post the dialog and wait for a user response
```

```

set answer = Alert (MUIObject, alertPropertiesList)
-- Dispose of the MUI Object
set MUIObject = 0
-- Check the answer against the possible user choices
put "User chose button" && answer
case (answer) of
  1: -- User chose first button (in this case 'Yes')
    halt -- the user chose to quit in this case
  2: -- User chose second button (in this case 'No')
    go frame 1 -- the user didn't quit, so start over
end case
end testMUIalert

```

Note that *Alert()* returns an integer indicating the number of the button that the user chose. For example, if the alert dialog box's buttons are *Yes* and *No*, a return value of 2 specifies that the second button (*No*) was pressed.

Table 21-1 shows each item used to define the alert dialog box. You need only specify the properties for which the default is inadequate, such as *#message* and *#buttons*.

---

---

Specifying the *#RetryCancel* option for *#buttons* may crash when creating the dialog in D7. I am not sure about its reliability in D7.0.2, or D6.5 and prior.

---

---

*Table 21-1: MUI Alert Properties*

Property	Usage	Range
<i>#buttons</i>	The buttons that appear in the alert (in the order in which they are named in each symbol, such as <i>#AbortRetryIgnore</i> )	<i>#AbortRetryIgnore</i> , <i>#Ok</i> (default), <i>#OkCancel</i> , <i>#RetryCancel</i> <i>#YesNo</i> , <i>#YesNoCancel</i>
<i>#default</i>	The ordinal number of the button to use as the default. <sup>1</sup> Specify 0 for no default.	0, 1, 2, or 3 (up to maximum number of buttons). Default is 1.
<i>#icon</i>	The type of icon that appears in the alert dialog. <sup>2</sup>	<i>#caution</i> , <i>#error</i> , <i>#note</i> , <i>#question</i> , <i>#stop</i> (default is no icon). See Figure 21-2.
<i>#message</i>	Message string that appears in the alert dialog.	Any string up to 255 characters (default is "<Null>" under Windows, and EMPTY on Mac)
<i>#movable</i> <sup>3</sup>	Indicates whether the dialog is moveable. <sup>4</sup>	TRUE or FALSE (default is FALSE on Mac and always TRUE under Windows).
<i>#title</i>	Title string for the alert dialog. <sup>4</sup>	Any string of up to 255 characters

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

		(default is "<Null>")
--	--	-----------------------

1. The default button choice is shown with a thick border and is selected if the user hits RETURN. If *#default* is omitted, the default is 1. For example, if *#buttons* is *#YesNoCancel*, the default choice would be the first button, in this case "Yes".
2. Omit the *#icon* property from the list to prevent an icon from being used. Specifying an invalid *#icon*, such as 0 or an invalid symbol, prevents the dialog from appearing and may cause a crash.
3. *#movable* is spelled without an "e" (differs from the *moveableSprite* of *sprite* property).
4. The dialog is always moveable and always contains a title bar under Windows, regardless of the *#movable* property. On the Macintosh, the *#title* is ignored (the alert box has no title) unless *#movable* is TRUE. If *#movable* is TRUE and *#title* is not specified, the title displays as "<Null>". Use *#title:EMPTY* to blank the title bar.

Note in Figure 21-2 that the appearance of the icons is platform-specific, and the same graphic is used for multiple *#icon* settings in some cases.

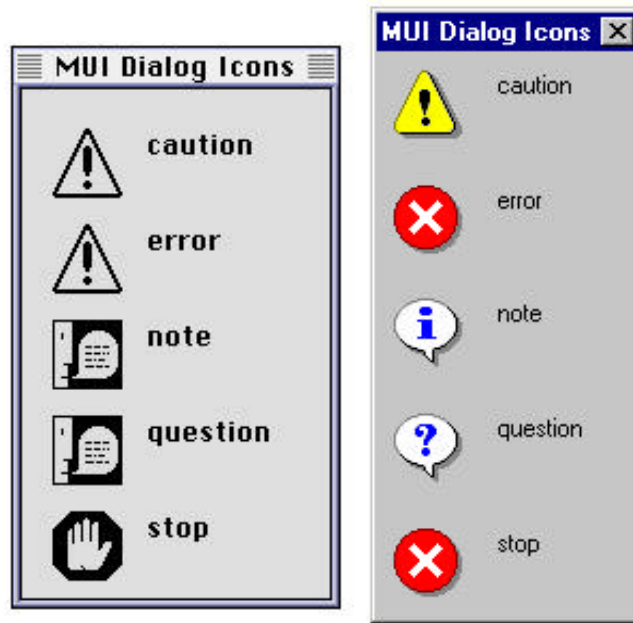


Figure 21-2: Macintosh and Windows MUI Dialog Icons

## Custom MUI Dialog Boxes

The MUI Xtra can create and control custom dialog boxes containing buttons, editable fields, labels, sliders, popup menus, and even bitmaps. It also creates simple alert dialogs as covered above, and file dialogs for saving and opening files or entering URLs as covered on pages 375-376 of Chapter 14, *External Files*, in *Lingo in a Nutshell*. (Note that the *FileOpen* method failed in the D6.0

version of the MUI Xtra, but it was fixed in the D6.0.2 release.)

MUI custom dialog boxes:

- Can contain numerous user interface elements
- Can include horizontal scrolling editable text fields
- Allow clipboard operations for editable text [doesn't seem to work in my tests?]
- Can be updated dynamically based on a user's choices.
- Use less memory than a MIAW used to simulate a dialog box
- Have the platform-specific appearance of native dialog boxes
- Select the default button when the user presses Enter or Return
- Can be canceled by pressing ESCAPE or Command-period.

MUI custom dialog boxes can contain very large text strings (more than 255 characters) but only if manual carriage returns are included in the string (every 100 characters or so) to break it onto multiple lines. Refer to the *#label* widget below.

## Making Sense of MUI

The MUI Xtra is more powerful than your feeble mind can comprehend (without my help). If you don't believe me, type this in the Message window:

```
| put mMessageList (xtra "MUI")
```

I'm convinced that *MUI* stands not for *Macromedia User Interface*, but for *Minimally Understandable Interface*, *Mostly Unintelligible Instructions*, or *Many Ungodly Items*.

---

You must understand Lingo lists *thoroughly* to have any  
prayer of using the MUI Xtra. See the *Lists* chapter.

---

The MUI Xtra is listed under "*MMUI Xtra*" (not *MUI Xtra*) in the D6 Help system index with links to related topics. The D6 for Macintosh CD has a very enlightening sample movie under *Director 6 CD:Goodies:Movies:dialogs.dir 1*. It is not included on the D6 for Windows CD or in the D6.0.2 updater, but you can download it from <ftp://ftp.shore.net/members/update/dialogs.zip>. It may also be included on the D6.5 updater CD under the "Goodies" folder.

UpdateStage has excellent discussions of the MUI Xtra and its inability to create dialogs bigger than the screen, a small sample Director movie, and a detailed MUI article by Macromedia's John C. Ware at:

<http://www.updatestage.com/previous/970801.html#item3>  
<ftp://ftp.shore.net/members/update/muihowto.zip>

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

*<ftp://ftp.shore.net/members/update/muilingo.zip>*

I owe John a debt of gratitude as his article clarified some points not covered in the on-line Help. It is getting a bit dated, and has some minor errors, but is very informative. There is no way I could possibly verify every feature and quirk of the MUI Xtra, but between John's article and the discussion that follows you should have an excellent foundation. When in doubt, you should trust your own instincts and perform your own tests, as it is certainly possible that I have made minor mistakes regarding the myriad properties. Furthermore, it may have changed in subsequent versions of the MUI Dialog Xtra (these tests were made with version 6.0.2r33 but I don't think major changes have been made in later versions).

---

The examples in this chapter, plus more, are available at the download site cited in the *Preface*.

---

Search Macromedia's site for the keywords, "MMUI", "MUI" and "MUIDialog" to find the relevant technotes pertaining to the MUI Xtra. User interface guidelines for laying out your own dialog boxes start at:

*<http://www.macromedia.com/support/xtras/how/mui/tips.html>*

If you don't limit the search to just the Director technotes, you'll find::

*[http://www.macromedia.com/support/xtras/xdks/xdk\\_d6a4/docs/html/mudg/index.htm](http://www.macromedia.com/support/xtras/xdks/xdk_d6a4/docs/html/mudg/index.htm)*.

(The MOA XDK documentation is also available in HTML format on the Director 6 CD under the *Macromedia/xdk\_d6a4* folder). Note that MUI's full power is available to MOA Xtras via the IMUIDialog interface, not all of which is exposed to Lingo (and some properties have different names).

Macromedia has posted a detailed white paper on the MUI Xtra at:

*<http://www.macromedia.com/support/director/how/d7/MUI.html>*

See also:

*<http://www.macromedia.com/support/director/how/subjects/usingxtras.html>*

UpdateStage publishes the MUI Maker Xtra by RavWare, which allows you to lay out your dialog visually and then writes the Lingo code for you. It does not create the callbacks for fancy custom dialogs, but presumably saves you layout time. I haven't tested it, but UpdateStage's products and support have a good reputation:

*<http://www.updatestage.com/xtras/muimaker.html>*

The free MUIComposer Xtra by Rainer Ohman may provide similar functionality, but I haven't tested it:

*<http://home1.swipnet.se/~w-10540/mui.htm>*

## The Life of a Dialog

Let's make sense of MUI in stages. To create and use a custom dialog we will:

1. Create an instance of the MUI Xtra to be used for subsequent operations.
2. Initialize the dialog by defining the window itself, and the user interface elements (widgets) within it. (Not to be confused with the Widget Wizard.
3. Display the dialog.
4. Detect user actions, and update the dialog in response.
5. Dismiss the dialog.

Tables 21-2 lists the MUI Xtra commands related to custom dialogs in the order in which they would ordinarily be used (excluding the *Alert()* function discussed earlier).

*Table 21-2: MUI Xtra Custom Dialog Commands.*

Command and Syntax	Usage
set <i>MUIobject</i> = new (xtra "MUI")	Instantiates the MUI Xtra.
<i>Alert(alertPropertiesList)</i>	Creates an alert dialog with one to three buttons (see above). Not used for custom dialogs.
<i>GetWindowPropList(MUIobject)</i>	Returns a default window property list used as basis for <i>#windowPropList</i> passed to <i>Initialize()</i> .
<i>GetWidgetList(MUIobject)</i>	Returns a list of currently supported widget types (see <i>#type</i> property in <i>GetItemPropList()</i> )
<i>GetItemPropList(MUIobject)</i>	Returns a default widget property list used as basis for elements in <i>#windowItemList</i> passed to <i>Initialize()</i> . <sup>1</sup>
<i>Initialize(MUIobject, [#windowPropList:windowProps, #windowItemList:windowItems])</i>	Defines (but does not display) a dialog and its contents. Dialog is displayed using <i>Run()</i> or <i>WindowOperation()</i>
<i>Run(MUIobject)</i>	Displays a modal dialog.
<i>Stop(MUIobject, 0)</i>	Dismisses a modal dialog.
<i>WindowOperation(MUIobject, #show   #hide)</i>	Shows or hides a non-modal dialog. (see Table 21-9).

ItemUpdate ( <i>MUIObject</i> , <i>itemNumber</i> , <i>itemInputPropList</i> )	Updates the specified item in the dialog box with the new properties.
FileOpen ( <i>MUIObject</i> , "defaultFile")	Displays File Open dialog. See the <i>External Files</i> chapter
FileSave ( <i>MUIObject</i> , <i>itemNumber</i> , <i>itemInputPropList</i> )	Displays File Save dialog. See the <i>External Files</i> chapter
GetURL ( <i>MUIObject</i> , url, <i>moveable</i> )	Displays URL dialog. See the <i>External Files</i> chapter
MoaErrorToString( <i>errCode</i> )	Translates MOA errors to text. See the <i>Lingo Coding and Debugging Tips</i> chapter and <i>Appendix E</i> .

1. *GetItemPropList()* would be more useful if it returned a default list specific to a given widget type. For some widgets, you must change most of returned list's values from the defaults.

Let's examine a degenerative example to make it all concrete before explaining each step in detail below (and I do explain it all!) The following example creates a simple dialog with one editable text field as shown in Figure 21-3.

---



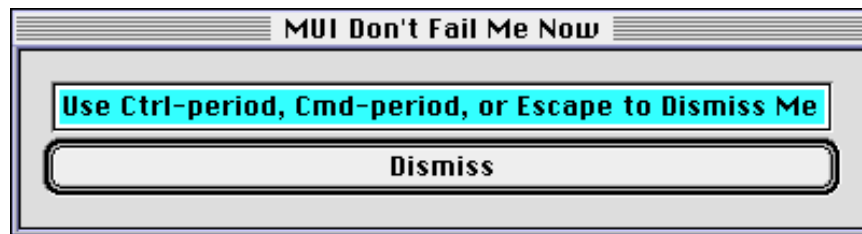
---

Use the dummy *dialogActions()* *callback handler* provided with this example. Without it, a modal dialog may hang your system. You have been warned. Use Command-period (Mac) Ctrl-period or Esc (Windows) to dismiss a dialog if stuck.

---



---



*Figure 21-3: Simple Custom Dialog*

The possible widget types for your dialog are shown in Table 21-7.

The first widget in a window must be *#WindowBegin* and the last widget must be *#WindowEnd*. Always create a separate property list for *each* widget using *GetItemPropList()*. Reusing a list reference will trample earlier widgets created with the same list, which can be quite confusing to debug.



**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

*Example 21-2: A Simple Custom MUI Dialog With An  
Editable Text Widget*

```
on simpleMUITest
  -- We'll want these for future use. Trust me!
  global gMUIObject
  global gWindowProps
  global gWindowItems

  -- Instantiate the Xtra
  set gMUIObject = new (xtra "MUI")

  -- Start with the default container window's attributes
  set gWindowProps = GetWindowPropList(gMUIObject)
  -- Set the title and width of the dialog
  set the name of gWindowProps = "MUI Don't Fail Me Now"
  set the width of gWindowProps = 250
  -- Specify callback handler that responds to user actions
  set the callback of gWindowProps = "dialogActions()"

  -- Build a list of the contents of the window
  set gWindowItems = []

  -- The first widget in the dialog must be #WindowBegin
  -- Start with a generic item's attributes
  set widget1 = GetItemPropList(gMUIObject)
  set the type of widget1 = #windowBegin
  add gWindowItems, widget1

  -- Add a simple text field
  set widget2 = GetItemPropList(gMUIObject)
  set the value of widget2 = ~
  "Use Ctrl-period, Cmd-period, or Escape to Dismiss Me"
  set the type of widget2 = #editText
  add gWindowItems, widget2

  -- Add a dismiss button
  set widget3 = GetItemPropList(gMUIObject)
  set the type of widget3 = #defaultPushButton
  set the title of widget3 = "Dismiss"
  add gWindowItems, widget3

  -- The last widget in the dialog must be #WindowEnd
  set widget4 = GetItemPropList(gMUIObject)
  set the type of widget4 = #windowEnd
  add gWindowItems, widget4

  -- Initialize the dialog
  Initialize (gMUIObject , [#windowPropList: gWindowProps, ~
  #windowItemList: gWindowItems])
  -- Draw the dialog
  Run (gMUIObject)
```

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

```
end simpleMUITest

-- This is your callback handler (see gWindowProps above)
on dialogActions event, itemNumber, itemPropList
    global gMUIObject
    put "Yippee! Received event" && event
    if not objectP(gMUIObject) then
        alert "gMUIObject must be declared globally elsewhere"
        exit
    end if
    case (event) of
        #itemClicked, #itemLosingFocus, #itemChanged:
            -- In an emergency this will stop both
            -- modal and non-modal dialogs.
            put "Dismissing dialog"
            stop(gMUIObject, 1)
            windowOperation (gMUIObject, #hide)
        otherwise:
            put "We don't handle that event yet:" && event
    end case
end dialogActions
```

The callback handler shown above is a degenerative example designed to avoid hanging your system. In a real example, it would respond to user actions (see *Interacting with a Dialog Box* far below) and would not dismiss our dialog unless the user hit the correct button.

## Instantiation

As with most Xtras, you must create an instance of the MUI Xtra using *new()*. You should store the instance in a global variable for future use.

*Example 21-3: Instantiating the MUI Xtra*

```
on instanceMUI
    global gMUIObject
    set gMUIObject = new (xtra "MUI")
    if not objectP(gMUIObject) then
        alert "Install the MUI Dialog Xtra and restart Director."
    end if
end instanceMUI
```

Each instance of the Xtra can pertain to a different dialog, allowing you to create and manage multiple concurrent custom dialogs.

## Setting up a Custom Dialog

To create the dialog we must define the appearance of its container window (it's size, position, etc.), and the attributes of one or more user interface elements within it (checkboxes, sliders, etc.) There are many attributes for the window itself and the widgets within it. Try not to get bogged down in the details of the

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

next few sections. Conceptually, we are simply defining the dialog window's attributes and storing them in a property list. We are then defining a complex list of the widgets (and their properties) that are placed within the dialog. Once the lists are defined, we'll initialize the dialog, using:

```
Initialize (gMUIObject, [#windowPropList: gWindowProps, ↵  
    #windowItemList: gWindowItems])
```

### Creating the Dialog Window

The attributes that determine a dialog window's appearance are specified in the `#windowPropList` property passed to `Initialize()`. The `GetWindowPropList()` function returns a generic list of the properties needed to define the window. You should use the value returned by `GetWindowPropList()` as the basis for your `#windowPropList` to ensure forward compatibility.

#### *Example 21-4: Creating the Window Property List*

```
put GetWindowPropList(gMUIObject)  
-- [#type: #normal, #name: "window", #callback: "nothing",  
    #mode: #data, #xPosition: 100, #yPosition: 120,  
    #width: 200, #height: 210, #modal: 1, #toolTips: 0,  
    #closeBox: 1, #canZoom: 0]
```

You can modify individual properties as follows:

```
set gWindowProps = GetWindowPropList(gMUIObject)  
set the name of    gWindowProps = "Nutshell Test"  
set the width of   gWindowProps = 250  
set the callback of gWindowProps = "dialogActions()"
```

(Do not be misled by the incorrect on-line Help link from the *Specifying overall dialog box properties* topic to the *GetItemPropList* topic. `GetItemPropList()` is used when specifying widgets *within* a dialog. `GetItemWindowList()` is used to define the dialog window itself).

If you create the window property list manually, you *must* include all the properties returned by `GetWindowPropList()` or your dialog will fail:

```
set gWindowProps = [#type: #alert, #name: "Nutshell Test", ↵  
    #callback: "dialogActions()", ↵  
    #mode: #data, #xPosition: -1, #yPosition: -1, ↵  
    #width: 0, #height: 0, #modal: TRUE, #toolTips: FALSE, ↵  
    #closeBox: FALSE, #canZoom: FALSE]
```

Table 21-3 lists the possible values for each property in the `#windowPropList`. The `#type`, `#name`, `#modal`, `#closeBox`, and `#canZoom` properties determine the appearance and modality of the window and its title bar. See Table 21-4 for details on the interaction between these properties, and their differences between Macintosh and Windows.

The `#xPosition`, `#yPosition`, `#width` and `#height` properties control the position and size of the dialog, but depend heavily on the `#mode` property (which also affects the layout of elements within the dialog). We'll revisit the dialog layout in *Dialog Coordinates* after discussing how to add elements to the dialog.

*Table 21-3: MUI #WindowPropList Elements*

Property	Usage	Default	Range
#type	Determines window type (see Table 21-4).	#normal	#alert, #normal, #palette
#name	Determines window's title.	"window"	Any string (Use EMPTY for no name)
#modal	Determines whether dialog is modal.	TRUE	TRUE or FALSE
#closeBox	Determines whether dialog has a close box.	TRUE	TRUE or FALSE
#canZoom	Determines whether dialog has a zoom box. <sup>1</sup>	FALSE	TRUE or FALSE
#callback	Lingo handler that is called whenever user interacts with dialog.	"nothing"	A handler name in quotes, such as " <i>dialogHandler</i> "
#toolTips	Determines whether to display tooltips. <sup>2</sup>	FALSE	N/A
#mode	Determines layout of the dialog (see Table 21-6)	#data	#data, #dialogUnit, #pixel
#xPosition	Offset from <i>top</i> of screen to <i>top</i> of dialog <sup>3</sup>	100	Positive or negative. (-1 centers dialog) <sup>1</sup>
#yPosition	Offset from <i>left</i> of screen to <i>left</i> of dialog <sup>3</sup>	120	Positive or negative. (-1 centers dialog) <sup>1</sup>
#height	Height of dialog, excluding title bar, if any.	210	1 to n (limited by screen height)
#width	Width of dialog	200	1 to n (limited by screen width)

1. The #canZoom property determines only whether a zoom box is displayed for the dialog. It won't actually zoom the window.

2. Tool tips are not yet supported as of version 6.0.2 of the MUI Dialog Xtra.

3. The #xPosition and #yPosition properties are reversed in version 6.0.2 of the MUI Dialog Xtra (see warning note below). Both #xPosition and #yPosition must be -1 in order to center the dialog. Negative coordinates, like large positive coordinates, can be used to place the dialog off-screen.

## Window Appearance

Table 21-4 details how the window's *#type* property changes its appearance and functionality on various platforms. These tests were performed using Windows 95 and Mac OS 7.5.5 during authoring. The results may vary slightly on Window 3.1, Windows NT or Mac OS 8, or from a Projector, but the table should illuminate the general relationships at play.

*Table 21-4: Dialog Appearance and Properties*

#type-->	#alert		#normal or #palette	
	Macintosh	Windows	Macintosh	Windows
<b>Usage</b>	Non-moveable dialog with no title.	Moveable dialog with title bar.	Moveable dialog with title bar.	Moveable dialog with title bar.
<b>#modal</b>	TRUE or FALSE	TRUE or FALSE	TRUE or FALSE	TRUE or FALSE
<b>#name<sup>1</sup></b>	Ignored (no title bar.)	Left-justified in title bar	Centered in title bar.	Left-justified in title bar
<b>#closeBox</b>	Ignored (no title bar.)	TRUE or FALSE	TRUE or FALSE (FALSE if #modal is TRUE)	TRUE or FALSE
<b>#canZoom</b>	Ignored (no title bar.)	Ignored	TRUE or FALSE	Ignored

Whew! All we've done so far is set up the container window's attributes. Now let's put something inside the dialog.

## Adding Items (Widgets) to the Dialog

The user interface elements (widgets) within the dialog are specified in the *#windowItemList* property passed to *Initialize()*. The *#windowItemList* is a *list of property lists* (one property list for each widget). The *GetItemPropList()* function returns a generic list of the properties that must be specified for each widget. You should use the value returned by *GetItemPropList()* as the basis for each widget's property list within *windowItemList* to ensure forward compatibility.

*Example 21-5: Defining a Single Widget's Properties*

```
| put GetItemPropList (gMUIobject)
```

```
-- [#value: 0, #type: #checkBox, #attributes: [],
#title: "title", #tip: "tip", #locH: 20, #locV: 24,
#width: 200, #height: 210, #enabled: 1]
```

You can modify individual properties as follows (see below for details) and then add it to the list of widgets we'll be building:

```
set itemList = []
set itemProps = GetItemPropList(gMUIObject)
set the type of itemProps = #radioButton
set the attributes of itemProps = [#textSize:#large, ~
  #textStyle:[#bold, #italic]]
add itemList, itemProps
```

If you create the item's property list manually, you *must* include all the properties returned by *GetItemPropList()*, for example:

```
set itemProps = [#value: 0, #type: #checkBox, ~
#attributes: [#textSize:#large, #textStyle:[#bold]], ~
#title: "Check Item", #tip: "tip", #locH: 20, #locV: 24, ~
#width: 0, #height: 0, #enabled: TRUE]
```

Table 21-5 lists the properties you must specify (via the *#windowItemList*) for each widget within the dialog. The *#attributes* property is covered in both Tables 21-7 and 21-8. The *#type* property must be one of the supported widget types as indicated by *GetWidgetList()*, and covered in Table 21-7.

*Table 21-5: MUI #WindowItemList Element Attributes*

Property	Usage	Default	Range
#type	Indicates widget type	#checkBox	See <i>GetWidgetList()</i> and Table 21-7.
#value	Initial value for widget	N/A	Depends on widget #type (see Table 21-7).
#title	Label used for widget.	"title"	Any string
#enabled	Determines whether widget is active. <sup>1</sup>	TRUE	TRUE or FALSE
#tip	Tool tip text for widget. <sup>2</sup>	"tip"	Any string or EMPTY
#attributes	Defines widget's text style, alignment, value range, and more.	[]	Depends on widget #type (see Tables 21-7 and 21-8).
#locH	Horizontal location of widget (offset from left edge of dialog). <sup>3</sup>	20	Any integer.

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

#locV	Vertical location of widget (offset from top edge of dialog). <sup>3</sup>	24	Any integer.
#width	Widget's width in pixels or dialog units. <sup>3</sup>	200	Any integer. (0 adjusts width automatically)
#height	Widget's height in pixels or dialog units. <sup>3</sup>	210	Any integer. (0 adjusts height automatically)

1. Disabled items appear dimmed in the dialog. Disabled buttons and sliders don't accept mouse input. Disabled editable fields don't accept keyboard input.

2. Tool tips are not yet supported as of version 6.0.2 of the MUI Dialog Xtra.

3. The #locH, #locV, #width, and #height are ignored if #mode:#data is specified in #windowPropList. They are measured in pixels if #mode is #pixel and dialog units if #mode is #dialogUnit.

## Dialog Coordinates and Element Positioning

The MUI Xtra offers both manual and automatic control of the window layout, as determined by the #mode property within the #windowPropList (see *Creating the Dialog Window* above). Table 21-6 shows the properties that control the position and size of the dialog and the widgets within it. Whether the units are pixels, or *dialog units* (see below), the upper left corner of the coordinate system is (0, 0). The dialog's position is specified relative to the monitor, and the widgets' positions are specified relative to the dialog. See Tables 21-3 and 21-5 above for details on each of the attributes.

---

The #xPosition and #yPosition properties are reversed (#xPosition determines the *vertical* offset and #yPosition determines the *horizontal* offset). (This is presumably a bug, and may be reversed in future versions). They are used to position the dialog in all layout modes, although the units may vary. Setting them *both* to -1 centers the dialog.

---

Don't confuse the *window's* properties with the *widgets'* properties, and remember that the window's #mode affects the units of measurement, the *window's size* and the *widgets' size and position*.

Table 21-6: Window and Widget Size and Position

#mode <sup>1</sup> -->	#data <sup>2</sup>	#pixel <sup>2</sup>	#dialogUnit <sup>3</sup>
Window's <i>horizontal</i>	#yPosition <sup>1</sup>	#yPosition <sup>1</sup>	#yPosition <sup>1</sup>

## DRAFT, August 3, 1999

*Lingo in a Nutshell*, published by O'Reilly & Associates

Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.

Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

position <sup>4</sup>			
Window's vertical position <sup>4</sup>	<i>#xPosition</i> <sup>1</sup>	<i>#xPosition</i> <sup>1</sup>	<i>#xPosition</i> <sup>1</sup>
Window's height	Sized automatically sized based on widgets' heights and number of widgets.	<i>#height</i> <sup>1</sup>	<i>#height</i> <sup>1</sup>
Window's width	Sized automatically sized based on widest widget or horizontal grouping.	<i>#width</i> <sup>1</sup>	<i>#width</i> <sup>1</sup>
Widget's position	Laid out automatically vertically from top to bottom, unless overridden (see below)	<i>#locH</i> and <i>#locV</i> <sup>5</sup>	<i>#locH</i> and <i>#locV</i> <sup>5</sup>
Widget's size	Each widget type assumes a default size, unless overridden (see below)	<i>#width</i> and <i>#height</i> <sup>5</sup>	<i>#width</i> and <i>#height</i> <sup>5</sup>

1. *#Mode*, *#xPosition* and *#yPosition* and the window's *#height* and *#width* are set in *#windowPropList* passed to *Initialize()*.

2. When *#mode* is *#data* or *#pixel*, all units are expressed in pixels.

3. When *#mode* is *#dialogUnit*, all units are expressed in dialog units, which depend on the system font and are roughly 1.5 times larger than pixels (i.e. 100 dialog units is approximately 150 pixels) using common default system fonts.

4. The *#xPosition* and *#yPosition* properties determine the offset from the upper left corner of the monitor in units that depends on the *#mode*. Note that the X and Y positions are reversed (see warning note above).

5. Each widget's *#locH*, *#locV*, *#height* and *#width* are specified in the *#windowItemList* passed to *Initialize()*.

### Dynamic Layout

The easiest approach is to use *dynamic layout mode* (*#mode:#data*) in which the height and width of the dialog *and the size and position of the widgets within it* are determined automatically as follows:

1. Items are drawn in the order in which they appear in the *#windowItemList* passed to *Initialize()*. The first widget appears at the top of the dialog, and the last one at the bottom, with each widget appearing on a separate line, unless overridden by a layout hint (see below).
2. A widget's *#locH*, and *#locV* properties are ignored, unless its *#attributes* include *[#layoutStyle:[#lockPosition]]*, in which case they override automatic positioning.



## DRAFT, August 3, 1999

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

3. Each widget is given a reasonable minimum height and width. It may be enlarged to fit its *#value* (for *#editText* widgets), or *#title* (for button widgets).
4. A widget's *#height* and *#width* properties are ignored, unless its *#attributes* include *[#layoutStyle:[#lockSize]]*, in which case they override automatic sizing.
5. Widgets may be resized to fit neatly within a horizontal or vertical group with other widgets. To quote John Ware: "Each subgroup is aligned and sized according to other items at its level in the hierarchy, regardless of whether the items are individual, or themselves groups." (see below). In fact, most widgets seem to be widened to match the widest widget.
6. The dialog is automatically sized to fit all the widgets including a reasonable border. The window's *#height* and *#width* properties ignored.

All items in a given vertical column will be given the same width. For example, if you have a wide editable text field, a button below it would be stretched very wide. In that case, you would have to alter the layout using layout hints (see below) or perform manual layout.

If the dialog's title is wider than the widgets, the dialog may not be wide enough to display the full title. In that case, either use a smaller title, larger widgets, or manual layout in which you can specify the *#width* of the dialog.

### Augmenting and Overriding Automatic Layout

When using dynamic layout mode, you'll often want to tweak the layout.

To align a *#label* widget, specify the text alignment (such as *[#textAlign:[#center]]*) as part of its *#attributes* property. The default tends to be left-alignment.

To specify a fixed size for a widget, include *[#layoutStyle:[#lockSize]]* in its *#attributes* property and set its *#height* and *#width* as desired. The default *#height* and *#width* properties are meaningless. (Buttons should generally be 20 pixels high). Set the *#height* and *#width* to zero to accept the "standard" size for the widget. *#LockSize* prevents it from being stretched to match other widgets.

To specify a fixed widget position, include *[#layoutStyle:[#lockPosition]]* in its *#attributes* property and set its *#locH*, and *#locV* as desired. (The default *#locH* and *#locV* are meaningless). Once you've used *#lockPosition* for a single widget, automatic positioning of subsequent widgets is unreliable.

### Layout Hints (Horizontal and Vertical Groups)

Each widget appears on a separate line of the dialog by default, which is often too simplistic. Fortunately, the *#GroupHBegin*, *#GroupHEnd*, *#GroupVBegin*, *#GroupVEnd* widgets provide so-called *layout hints*, that create horizontal and vertical groups of items, which can be nested (see below). Also refer to the

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

`#dividerH` and `#dividerV` widgets that are cosmetic only.

Items within groups will be sized automatically to provide more consistent appearance or uniform justification.

Use the `#GroupHBegin`, `#GroupHEnd` widgets to create a horizontal grouping of more than one widget (such as `#label` and `#editText`) on the same line of a dialog. Use the `#GroupVBegin`, `#GroupVEnd` widgets to create a vertical grouping of more than one widget (such as several buttons) that are then incorporated as a unit into a horizontal grouping.

*Example 21-6: Example Widget Layout*

For example, to create a layout similar to the one in Figure 21-4, use the following widgets:

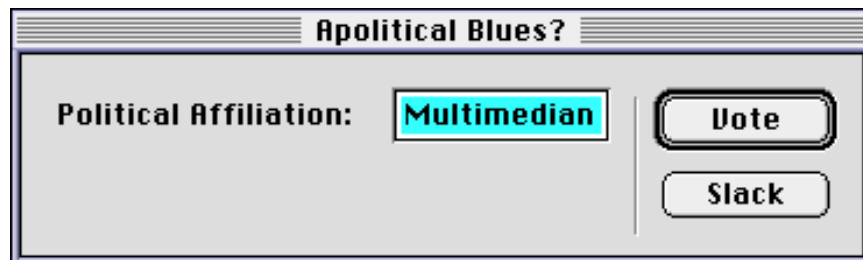
```
#windowBegin
  #groupHBegin

    #groupHBegin
      #label
      #editText
    #groupHEnd

    #dividerV

    #groupVBegin
      #defaultPushButton
      #cancelPushButton
    #groupVEnd

  #groupHEnd
#windowEnd
```



*Figure 21-4: Grouping Widgets in a Dialog*

Layout hints are intended solely for dynamic layout mode (`#mode:#data`). In *dynamic layout* mode, if two vertical groups are adjacent within a horizontal group, and the first group contains *only* `#label` widgets, these labels are automatically lined up with the items in the second vertical group.

Radio buttons inside of any group automatically become mutually exclusive regardless of the layout modes.

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

## Manual Layout

To manually specify the absolute size of the dialog and the size and position of widgets within it, use *#mode:#pixel*. You can use *#mode:#dialogUnit*. (particularly under Windows for the *Large Fonts* Display setting) to adjust the layout based on the user's system font size setting. Either manual mode requires substantially more effort than automatic layout, but gives you much greater control.

*Dialog units* are based on the current system font's height and width. If the system font changes, the number of pixels represented by a single dialog unit changes. When using dialog units, test your dialog's layout with different size fonts on both platforms. The system font size is set using the *Display Control Panel's Settings* tab under Windows (to set either the small or large font size), and using the *Views Control Panel* on the Macintosh. Dialog units tend to be roughly 1.4 to 1.9 times bigger than pixel units, but the conversion factor varies with the font and point size and also differs in the horizontal and vertical axes. Under Windows 95 using *Small Fonts*, a 320 by 240 *dialog units* window spans 480 by 385 *pixels*. On the Macintosh, the same dialog would span 450 by 365 pixels when using 9-point Geneva, and 600 by 420 pixels when using 18-point Geneva as the system font.

When the window's *#mode* is *#dialogUnit* or *#pixel*, setting the *window's #height* or *#width* to zero creates a zero-sized dialog. But, setting a *widget's #height* or *#width* to zero sizes that widget automatically, as would occur by default in dynamic layout mode.

## Widget-Specific Properties and Events

There are 20 possible widget types, with more expected in the future. A list of currently supported widget types can be obtained from *GetWidgetList()*:

*Example 21-7: Supported Widget List*

```
set gMUIobject = new (extra "MUI")
put GetWidgetList(gMUIobject)
-- [#dividerV, #dividerH, #bitmap, #checkBox, #radioButton,
#PopupMenu, #editText, #WindowBegin, #WindowEnd,
#GroupHBegin, #GroupHEnd, #GroupVBegin, #GroupVEnd, #label,
#IntegerSliderH, #FloatSliderH, #defaultPushButton,
#cancelPushButton, #pushButton, #toggleButton]
```

Most widgets have their specific attributes and quirks:

The *#dividerH* and *#dividerV* widgets add horizontal and vertical lines. *#DividerH* is often used to set off a vertical group defined by *#GroupVBegin* and *#GroupVEnd*. *#DividerV* is often used to set off a horizontal group defined by *#GroupHBegin* and *#GroupHEnd*.

Multiple *#radioButton* widgets within a group are automatically treated as mutually exclusive choices.

The *#defaultPushButton* widget has an OS-specific appearance indicating

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates

Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.

Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

that it is the default choice. You should only have one *#defaultPushButton*. If a *#defaultPushButton* widget is present, the RETURN key sends an *#itemClicked* event to the callback handler specifying its widget number.

You should only have one *#cancelPushButton*. If a *#cancelPushButton* widget is present, the Escape sends an *#itemClicked* event to the callback handler specifying the cancel button's widget number (but does *not* dismiss the dialog). If no *#cancelPushButton* widget is present, the Escape key dismisses the dialog and sends a *#windowClosed* event instead.

*#PushButton* widgets always return to their initial state, but *#toggleButton* widgets alternate between their toggled and untoggled states. The state of a toggle button (or check box or radio button) is indicated by its *#value* property (TRUE or FALSE).

Table 21-7 shows each widget type and its widget-specific values for the *#value* and *#attributes* properties. If there is a bullet in the *#title* column, the widget's title is displayed. Otherwise, you'll need a separate *#label* widget. If there is a bullet in the *#enabled* column, the widget can be enabled and disabled. Disabled widgets don't respond to mouse clicks or keyboard input. Note that the *#layoutStyle* attribute applies to all widgets except those used for layout hints. See Table 21-6 for details on the *#width*, *#height*, *#locH* and *#locV* widget properties. See *The #Attributes Widget Property* below for details on the unnervingly complex *#attributes* widget. See Table 21-10 for details on the events issued when various widgets are clicked or the user hits a key.

*Table 21-7: Widget-specific Attributes and Events*

#type	#value	#attributes	#title	#enabled	Events
#bitmap	cast member reference <sup>1</sup>	#layoutStyle, #bitmapIcon			#itemClicked, #itemChanged <sup>w</sup>
#checkBox	TRUE   FALSE	#layoutStyle, #textSize	•	•	#itemChanged, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w</sup>
#radioButton	TRUE   FALSE	#layoutStyle, #textSize	•	•	#itemChanged, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w,7</sup>
#toggleButton <sup>2</sup>	TRUE   FALSE	#layoutStyle, #textSize	•	•	#itemChanged, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w</sup>

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

#defaultPush Button	N/A (see #title)	#layoutStyle, #textSize	•	•	#itemClicked, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w</sup>
#cancelPush Button	N/A (see #title)	#layoutStyle, #textSize	•	•	#itemClicked, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w</sup>
#pushButton	N/A (see #title)	#layoutStyle, #textSize	•	•	#itemClicked, #itemEnteringFocus <sup>w</sup> , #itemLosingFocus <sup>w</sup>
#PopupList	String, integer or float	#layoutStyle, #popupStyle, #valueList <sup>3</sup>		•	#itemChanged, #itemEnteringFocus <sup>w,6</sup> , #itemLosingFocus <sup>w</sup>
#editText	Any string	#layoutStyle, #textSize, #textAlign, #textStyle		•	#itemChanged, #itemEnteringFocus, #itemLosingFocus
#label	Any String <sup>4</sup>	#layoutStyle, #textSize, #textAlign, #textStyle		•	#itemClicked <sup>m</sup>
#IntegerSliderH	Any integer	#layoutStyle, #sliderStyle, #valueRange <sup>3</sup>		•	#itemChanged
#FloatSliderH	Any float <sup>5</sup>	#layoutStyle, #sliderStyle, #valueRange <sup>3</sup>		•	#itemChanged <sup>5</sup>
#dividerH	N/A	#layoutStyle			#itemChanged <sup>m</sup>
#dividerV	N/A	#layoutStyle			#itemChanged <sup>m</sup>
#WindowBegin	N/A	N/A			
#WindowEnd	N/A	N/A			
#GroupHBegin	N/A	N/A			
#GroupHEnd	N/A	N/A			

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

#GroupVBeg in	N/A	N/A			
#GroupVEnd	N/A	N/A			

M. Receives this event on the Macintosh only.

W. Receives this event under Windows only.

1. The *#value* is mandatory for a *#bitmap* widget, unless the *#bitmapIcon* attribute is set, (The *#bitmapIcon* overrides the *bitmap #value* if both are specified).
2. The initial display of a *#toggleButton* with its *#value* set to TRUE is incorrect, although the internal *#value* is reported correctly. The button will correct itself if clicked.
3. The *#valueList* and *#valueRange* are mandatory. Director may crash if they are not specified for pop-up or slider widgets.
4. The *#label* widget's string can not be wider than the screen. Include manual carriage returns to wrap the string onto multiple lines.
5. If the initial value for a *#FloatSliderH* widget is not a float, the slider returns only *integer* values. Use a floating-point initial value, such as 0.0 or 10.0.
6. Under Windows, the *#PopupList* widget receives incorrect events when it gains or loses focus. It seems to receive spurious *#itemLosingFocus* events when making a menu selection, and receive an *#itemEnteringFocus* event rather than *#itemLosingFocus* when truly losing focus..
7. The *#radioButton* widget gains and loses focus with mouse clicks, but not with the Tab key as other widgets do under Windows. Neither does it toggle with the space bar as is typical for Windows radio buttons.

### **The #Attributes Widget Property**

The *#attributes* widget property is somewhat complex. It is a property list, allowing Macromedia to add more properties in the future. Some of it's subproperties are themselves linear lists or property lists, again allowing for future expansion.

Table 21-8 provides details on the possible values for each existing subproperty within the *#attributes* property. Items in the table separated by "|" indicate a range of exclusive choices. Items shown in a list can be specified in unison.

#### *Example 21-8: Setting Attributes of a Single Widget*

For example, the following could specify the *#valueList* attribute for a *#PopupList* widget:

```
set itemProps = GetItemPropList(gMUIobject)
set the type of itemProps = #PopupList
set the attributes of itemProps = ~
    [#valueList: ["choice1", "choice2", "choice3"]]
```

And the following could set up the *#textStyle*, *#textAlign* and *#textSize* attributes for an *#editText* widget.

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

```
set itemProps = GetItemPropList(gMUIObject)
set the type of itemProps = #editText
set the attributes of itemProps = [ ↵
    #textStyle: [#bold, #italic], ↵
    #textAlign: [#center], ↵
    #textSize: #large]
```

Here we have used the `#layoutStyle:[#lockSize]` attribute to override the default widget size using `#height` and `#width`. And we have used the `#layoutStyle:[#lockPosition]` attribute to override the default widget position using `#locH`, and `#locV`:

```
set itemProps = GetItemPropList(gMUIObject)
set the type of itemProps = #editText
set the attributes of itemProps = [ ↵
    #layoutStyle: [#lockSize, #lockPosition]]
set the height of itemProps = 75
set the width of itemProps = 100
set the locH of itemProps = 200
set the locV of itemProps = 250
```

*Table 21-8: Sub-Properties for #Attributes Property*

#attributes	Default
#textSize: #large   #tiny   #normal	#textSize: #normal
#textStyle: [#bold, #italic, #underline, #plain, #inverse] <sup>1</sup>	#textStyle:[#plain]
#textAlign: #left   #right   #center	system language standard
#popupStyle: #tiny   #cramped   #normal	#popupStyle:#normal
#valueList: ["string1", "string2", "string3"]	None. Must be specified for #popupList widgets
#valueRange:[#min: <i>min</i> , #max: <i>max</i> , #increment: <i>incr</i> , #jump: <i>jump</i> , #acceleration: <i>acc</i> ]	None. Must be specified for #IntegerSliderH and #FloatSliderH widgets.
#sliderStyle: [#ticks, #value] <sup>2</sup>	#sliderStyle:[]
#layoutStyle: [#lockPosition, #lockSize] <sup>3</sup>	#layoutStyle: [] (accept defaults)
#bitmapIcon: [#stop   #note   #caution   #question   #error] <sup>4</sup>	None

1. The `#inverse` text style is not yet supported as of version 6.0.2 of the MUI Dialog Xtra.

2. `#Ticks` includes tick *marks* along the slider and has nothing to do with the ticks used to measure time. `#Value` adds a set of arrow buttons that change the slider position by the

value specified by the `#valueRange`'s `#increment` property.

3. Macromedia's documentation claims that the `#layoutStyle` attribute also supports `#minimize`, `#centerH`, `#centerV`, `#right`, `#left`, `#top`, `#bottom`, but they don't seem to work.

4. The attribute's name is `#bitmapIcon` not `#bitmapStyle` as alluded to in the on-line Help. The example *set the attributes of tempPropItemList = [#bitmapIcon:#caution]* is correct. See Figures 21-2a and 21-2b for the appearance of the various icons on each platform.

Whew! We've covered all the properties for the dialog window and the widgets within it, but we haven't even begun to use the dialog.

## Window Layout Caveats and Tips

You can crash your system or get stuck in a modal dialog that you can not dismiss if you are not careful.

---

---

Save your work frequently when testing MUI. Try using  
Escape or Command-period if you are stuck in a dialog.

---

---

Refer to *Interacting with a Dialog* below for a minimalist *callback* handler that will guarantee that you can dismiss your dialog.

The following tips should help when creating the `#windowPropList`:

The `#windowPropList` is a property list. You must specify all of its properties or it won't work. Use `GetWindowPropList()` as the basis for your `#windowPropList` to ensure forward compatibility.

Using `#mode:#dialogUnit` or `#mode:#pixel` gives you complete control over dialog layout (see below). Use `#mode:#data` to have the MUI Xtra automatically calculate the size of the window based on the size and position of the widgets.

Avoid setting `#xPosition`, `#yPosition`, `#height` and `#width` such that the dialog is off-screen, resulting in an error or an invisible dialog.

The `#xPosition` and `#yPosition` are reversed in version 6.0.2r33 of the MUI Xtra (on both Macintosh and Windows)

Set *both* the `#xPosition` and `#yPosition` to -1 to center the dialog.

When using `#mode:#dialogUnit`, the units vary with the system font, and differ in the horizontal and vertical directions. Dialog units are usually larger than pixels.

The following tips should help when creating the `#windowItemList`:

The `#windowItemList` is a list of property lists. `GetItemPropList()` returns a sample item property list. You must add one such list to the `#windowItemList` for *each* widget (i.e. item). Use `GetItemPropList()` as the basis for each item in your `#windowItemList` to ensure forward compatibility.

You must retrieve a separate copy of the widget properties list using



**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates

Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.

Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

*GetItemPropList()* for each widget. If you reuse the same property list for multiple widgets, their widget attributes will trample all over each other.

Be sure to use *#WindowBegin* and *#WindowEnd* exactly once as the first and last widgets respectively in the *#windowItemList*.

Use *#mode:#data* to calculate the size and position of the widgets automatically.

Don't specify more widgets than fit on the user's screen. Group widgets to include more than one widget on a line.

When using *#mode:#dialogUnit* or *#mode:#pixel* layout, be careful not to:

- Create widgets that overlap (unless you do so intentionally)

- Position widgets outside the limits of the dialog.

- Position the dialog off-screen.

- Making widgets too small to be visible. Set the widget's *#height* and *#width* (not the window's *#height* and *#width*) to zero to size the widget automatically.

- Make widgets too big. The default widget's *#height* and *#width* creates comically large buttons.

If the dialog is not visible, check the following:

- Specifying an invalid symbol for the window's *#mode* property prevents the dialog from being displayed. Valid symbols are *#data*, *#pixel* and *#dialogUnit* (not *#pixels* or *#dialogUnits*).

- Using *VOID* as the window's *#name* property prevents the dialog from being displayed. Use *EMPTY* (or window *#type:#alert* on the Macintosh) to hide the name.

- If the size of the dialog (specified either manually or calculated automatically) is too big, it won't appear. Test on a the target platform which may have a smaller monitor than your development machine. On a 640-by-480 screen you can only have about 15 vertically arranged elements before the dialog exceeds the screen size. Add horizontal groups to use space more economically.

- Long *#label* widgets (greater than about 100 characters) will overflow the dialog's width limit and prevent it from being displayed. Include hard carriage returns in the string so that no one line is too long.

- A dialog isn't displayed by the *Initialize()* command. Use *Run()* to display modal dialogs and *WindowOperation(#show)* to display non-modal dialogs. (See the *#modal* window property).

Other Caveats:

- Always provide a way to dismiss the dialog, usually via a *#cancelPushButton*, *#defaultPushButton* or *#pushButton* widget with corresponding code in the callback handler to dismiss the dialog.

- Use *Stop(MUIObject, stopCode)* to dismiss a modal dialog. Use any integer as the *stopCode* (even though it is ignored) or it will fail.

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

Use *WindowOperation(#hide)* (not *WindowOperation(#stop)*) to dismiss a non-modal dialog. (See the *#modal* window property).

*#GroupHBegin* and *#GroupHEnd*, or *#GroupVBegin* and *#GroupVEnd* must be used in matching pairs, and can be nested, but not overlapped.

Correct:

```
| #GroupHBegin  
|   #GroupVBegin  
|   #GroupVEnd  
| #GroupHEnd
```

Wrong:

```
| #GroupHBegin  
|   #GroupVBegin  
| #GroupHEnd  
| #GroupVEnd
```

## Running and Interacting with the Dialog

Once the dialog has been defined with *Initialize()*, we are only half-way to our goal.

*Example 21-9: The Life and Times of a MUI Dialog*

To recap the entire process:

1. Instantiate the MUI Xtra, using *set MUIObject = new (xtra "MUI")*
2. Define the window and its widgets and then create the dialog using:

```
| Initialize(MUIObject, [#windowPropList: windowProps, -  
|   #windowItemList:windowItems])
```
3. Open a modal dialog using *Run(MUIObject)*. Open a non-modal dialog using *WindowOperation(MUIObject, #show)*.
4. User actions are trapped by the handler specified in the *#callback* property of the *#windowPropList* passed to *Initialize()*.
5. Use *ItemUpdate()* to update the dialog's appearance in response to user actions (see below).
6. Close a modal dialog using *Stop(MUIObject, 0)*. Close a non-modal dialog using *WindowOperation(MUIObject, #hide)*.
7. Dispose of the *MUIObject* by setting it to zero.

### Showing and Hiding Dialogs

The manner in which to open and close the dialog depends on whether it is modal or non-modal. In the following examples, it is assumed that you have already initialized the dialog using the *initializeMUI* handler in Example 21-10.

*Example 21-10: Initializing the MUI Dialog:*

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

```
on initializeMUI
  global gMUIObject, gWindowProps, gWindowItems
  set gMUIObject = new (xtra "MUI")
  if objectP(gMUIObject) then
    Initialize (gMUIObject, [#windowPropList: gWindowProps, ↵
      #windowItemList: gWindowItems])
  else
    alert "MUI Xtra doesn't seem to be installed"
  end if
end initializeMUI
```

Open a *modal* dialog box using:

```
| Run(gMUIObject)
```

Dismiss a *modal* dialog box using

```
| Stop(gMUIObject, 0)
```

The *WindowOperation()* command is used to show and hide *non-modal* dialogs.  
It takes the form:

```
| WindowOperation(gMUIObject, #action)
```

where *#action* is one of the properties shown in Table 21-9, such as:

```
-- Display a non-modal dialog
WindowOperation(gMUIObject, #show)
-- Dismiss a non-modal dialog
WindowOperation(gMUIObject, #hide)
```

*Table 21-9: WindowOperation() Actions*

Action	Usage
#show	Displays a non-modal dialog box only.
#hide	Hides a non-modal dialog box.
#zoom	Sends the #windowZoomed message to the #callback handler. Nothing happens unless the callback handler implements a zoom manually.
#center <sup>1</sup>	Centers the window on the monitor screen.
#tipsOn <sup>1</sup>	Turns tool tips on.
#tipsOff <sup>1</sup>	Turns tool tips off.

1. The #tipsOn and #tipsOff actions are not yet supported as of version 6.0.2 of the MUI Dialog Xtra. The #center does not work on either platform in my tests.

Assuming you've stored the #windowPropList in the global variable *gWindowProps*, you can use the following utilities to open and close the dialog without regard to whether it is modal.

*Example 21-11: Running and Dismissing Modal and Non-*

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

*Modal MUI Dialogs:*

```
on runDialog whichDialog, whichProps
  global gMUIObject
  global gWindowProps

  if voidP(whichDialog) then
    set whichDialog = gMUIObject
  end if

  if not objectP(whichDialog) then
    put "Not a valid dialog instance:" && whichDialog
    exit
  end if

  if voidP(whichProps) then
    set whichProps = gWindowProps
  end if

  if listP(whichProps) then
    if (the modal of whichProps) then
      Run(whichDialog)
    else
      WindowOperation (whichDialog, #show)
    end if
  else
    -- We don't know if it is modal, so let's try both
    Run(whichDialog)
    windowOperation (whichDialog, #show)
  end if
end runDialog

on stopDialog whichDialog, whichProps
  global gMUIObject
  global gWindowProps

  if voidP(whichDialog) then
    set whichDialog = gMUIObject
  end if

  if not objectP(whichDialog) then
    put "Not a valid dialog instance:" && whichDialog
    exit
  end if

  if voidP(whichProps) then
    set whichProps = gWindowProps
  end if

  if listP(whichProps) then
    if (the modal of whichProps) then
      Stop(whichDialog, 1)
    end if
  end if
end stopDialog
```

```
else
    WindowOperation (whichDialog, #hide)
end if
else
    -- We don't know if it is modal, so let's try both
    Stop(whichDialog, 1)
    windowOperation (whichDialog, #hide)
end if
end stopDialog
```

## Interacting with a Dialog Box

The MUI Dialog Xtra automatically handles basic interactivity for each widget in the dialog without additional programming. Keystrokes automatically register in editable text fields, sliders and pop-up menus react to the mouse or keyboard, buttons can be highlighted, clicked, etc. But to determine the value of each widget in the dialog or modify the dialog's contents in response to user actions you must create a *callback handler* that determines which widget was clicked and takes appropriate action. Don't get confused by the name *callback handler*. It is just a handler that is called each time the user does something of note in the dialog.

Creating interaction is a three-step process:

1. Set the *#callback* property (see below)
2. Write the callback handler (see *Detecting User Actions* below)
3. Use *ItemUpdate()* from within the callback handler.

Set up the callback handler using the *#callback* property in the *#windowPropList* passed to *Initialize()*, as shown eons ago under *The Life of a Dialog..* Here is a small excerpt.

### Example 21-12: MUI Lifetime Recap

```
global gMUIobject
global gWindowProps
global gWindowItems
set gMUIobject = new (xtra "MUI")
set gWindowProps = GetWindowPropList(gMUIobject)
-- Specify the handler that respond to user actions
set the callback of gWindowProps = "dialogActions()"
-- Set up the gWindowItems item list (not shown)
Initialize (gMUIobject, [#windowPropList: gWindowProps, ~
    #windowItemList: gWindowItems])
Run(gMUIobject) -- Or WindowOperation for non-modal dialogs
```

## Detecting User Actions

When anything happens in the dialog box, the callback handler is called. Your callback handler will depend entirely on the purpose of each widget in your

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates

Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.

Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

dialog. Once you've written your callback handler (see below) it should be placed in a movie script. Use this test callback handler to get your feet wet. It assumes that the global *gMUIobject* is the instance of the MUI Xtra you used to initialize and display your dialog. It dismisses the dialog when any button within the dialog is clicked (which is not very useful but ensures that you don't hang your system with a modal dialog that can't be closed).

*Example 21-13: Sample Callback Handler*

```
on dialogActions event, itemNumber, itemPropList
  global gMUIobject
  put "Yippee! Received event" && event
  if not objectP(gMUIobject) then
    alert "gMUIobject must be declared globally elsewhere"
    exit
  end if
  case (event) of
    #itemClicked:
      -- In an emergency this will stop both
      -- modal and non-modal dialogs.
      stop(gMUIobject, 1)
      windowOperation (gMUIobject, #hide)
    otherwise:
      put "We don't handle that event yet:"
  end case
end
```

---

---

Test with a non-modal dialog to avoid freezing your system until you get the callback handler working. Always provide some way to dismiss the dialog in your callback handler.

---

---

The callback handler receives up to three parameters describing the event:

The first parameter (*event*) is always a symbol indicating the *event* as shown in Table 21-10.

If *event* is item-related (*#itemChanged*, *#itemClicked*, *#itemEnteringFocus*, or *#itemLosingFocus*), the second parameter (*itemNumber*) indicates the item's position in the original *#windowItemList* passed to *Initialize()*.

If the event is item-related, the third parameter (*itemPropList*) contains the current item property list for the affected item.

If first parameter (*event*) is *#windowClosed*, *#windowOpening*, *#windowResized*, or *#windowZoomed*, then the second and third parameters are VOID. Don't assume that the third parameter (*itemPropList*) is always a list.

Widgets generate either an *#itemClicked* or *#itemChanged* event, not both, although widgets also generate *#itemEnteringFocus* and *#itemLosingFocus*. See below for details on how to use this information in your callback handler.

*Table 21-10: Callback Events*

Event	Usage	Issued by
#itemChanged <sup>1</sup>	Sent when an item has changed. <sup>2</sup>	#checkBox, #radioButton, #toggleButton, #popupList, #editText, #IntegerSliderH, #FloatSliderH, #dividerH <sup>m</sup> , #dividerV <sup>m</sup>
#itemClicked <sup>1</sup>	A button or label has been clicked. <sup>3</sup>	#bitmap, #defaultPushButton, <sup>3</sup> #cancelPushButton, <sup>3</sup> #pushButton, #label <sup>m</sup>
#itemEnteringFocus <sup>1</sup>	An editable text field gained focus. <sup>4</sup>	#editText, #checkBox <sup>w</sup> , #radioButton <sup>w</sup> , #toggleButton <sup>w</sup> , #popupList <sup>w</sup> , #defaultPushButton <sup>w</sup> , #cancelPushButton <sup>w</sup> , #pushButton <sup>w</sup>
#itemLosingFocus <sup>1</sup>	An editable text field lost focus	#editText, #checkBox <sup>w</sup> , #radioButton <sup>w</sup> , #toggleButton <sup>w</sup> , #popupList <sup>w</sup> , #defaultPushButton <sup>w</sup> , #cancelPushButton <sup>w</sup> , #pushButton <sup>w</sup>
#windowClosed	Dialog was closed. <sup>5</sup>	The close box, <i>Stop()</i> or <i>WindowOperation(#hide)</i> . <sup>5</sup>
#windowOpening	Dialog was opened.	<i>Run()</i> or <i>WindowOperation(#show)</i> .
#windowResized	Dialog was resized (not yet supported).	N/A
#windowZoomed	Dialog zoom was requested. <sup>6</sup>	Zoom box, or <i>WindowOperation(#zoom)</i>

m. Macintosh-only

w. Windows-only

1. Callback handler receives item's number and property list as parameters following the item-related events.

2. The *#itemChanged* event is sent when

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

A keystroke affects the contents of an editable text field.

A *#FloatSliderH* or *#IntegerSliderH* widget's value changes (i.e. only if the slider moves, not when it gains focus) whether using the arrow keys (Windows only) or the mouse.

The user re-selects the current menu choice from a *#PopupMenu*.

User clicks on a *#dividerH* or *#dividerV* widget!

3. The *#itemClicked* event is also sent when:

The RETURN key is pressed on the Macintosh (if a *#defaultPushButton* widget is defined),  
The ESCAPE key is pressed on the Macintosh (if a *#cancelPushButton* widget is defined).  
The ENTER key is pressed under Windows when any type of push button has focus under Windows. The SPACE bar is pressed under Windows when a radio button, check box or toggle button has focus.

4. An *#editText* widget can lose focus to another *#editText* widget, or to a *#defaultPushButton* when the RETURN key was pressed.

5. The *#windowClosed* event is also sent when the ESCAPE key is pressed, but only if a *#cancelPushButton* widget is *not* defined.

6. Nothing happens unless you resize the dialog manually in your callback handler, which appears to be possible only if you re-initialize the entire dialog.

### An Improved Callback Handler

Now that we have a grasp on how callback handlers can decipher different events, let's look at an improved version of our callback handler from Example 21-13 in Example 21-14. This can replace the earlier callback handler once you have gotten the dialog layout as desired, and want to add some functionality. It is merely a skeleton. You'll have to add whatever you want for your particular dialog. See the example Director movies cited at the beginning of this section for some examples.

Note that for *#itemChanged* events we inspect the item's *#value* to determine the widget's current state or contents. For *#itemClicked* events, we inspect the item's title or number to determine which button was clicked. In either case, *the type of widgetProps* tells us how to interpret the item.

#### Example 21-14: Complete Callback Handler Template

```
on dialogActions event, widgeNum, widgeProps
    global gMUIObject
    global gWindowProps
    global gWindowItems

    if voidP(widgeNum) then
        put "Window event detected:" && event
    else
        put "Widget" && widgeNum && "generated:" && event
        -- Determine the widget generating the event
        set widgeType = the type of widgeProps
        set widgeName = the title of widgeProps
        set widgeValue = the value of widgeProps
```



```
end if

case (event) of
  #itemEnteringFocus, #itemLosingFocus:
    -- You'll usually ignore these events generated
    -- for #editText widgets (and others under Windows)

  #itemChanged:
    -- One of the following widget types has changed:
    -- #checkBox, #radioButton, #toggleButton,
    -- #popupList, #editText, #IntegerSliderH,
    -- #FloatSliderH (#dividerH and #dividerV on Mac)
    case (widgetType) of
      #checkBox, #radioButton, #toggleButton:
        put "State of" && widgetName && "is:" && widgetValue

      #popupList:
        put "Picked" && widgetValue && "from" && widgetName

      #editText:
        -- This is called each time a keystroke
        -- affects the contents of a text field.
        -- Filter out characters using ItemUpdate()
        put "The last key pressed was" && the key
        put "Field" && widgetName && ":" && widgetValue
        -- Disallow any spaces, for example:
        if (the key) = SPACE then
          beep
          -- Reset previous contents of text field
          -- That we've kept stored on a running basis
          if listP (gWindowItems) then
            set the value of widgetProps = ~
              the value of getAt (gWindowItems, widgetNum)
          end if
          if objectP (gMUIObject) then
            ItemUpdate (gMUIObject, widgetNum, widgetProps)
          else
            alert "gMUIObject should be a MUI instance"
          end if
        end if

      #IntegerSliderH, #FloatSliderH:
        put "Slider" && widgetName && widgetValue

      #dividerH, #dividerV:
        nothing
    end case

  -- Store the new value of the widget
  -- back into our global widget property list
  if listP (gWindowItems) then
    setAt (gWindowItems, widgetNum, widgetProps)
```

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

```
else
    put "gWindowItems should be #windowItemList"
end if

#itemClicked:
-- One of these widgets has been clicked:
-- #defaultPushButton, #cancelPushButton,
-- #pushButton, #bitmap (or #label on Mac)
put widgetName && widgetType && "clicked"
case (widgetType) of
    #defaultPushButton :
        -- There should be only one #defaultPushButton
        -- If default is OK, commit any changes
        -- If default is Abort, revert any changes
        -- Either way, you should dismiss the dialog.
        stopDialog()
    #cancelPushButton:
        -- There should be only one #cancelPushButton
        -- Abort any changes and then dismiss the dialog
        stopDialog()
    #pushButton:
        -- Use #title to determine which button was pushed
        -- or use widgetNum if button names are not unique.
        case (widgetName) of
            "OK":
                -- If OK button was not a #defaultPushButton,
                --- you can handle it here.
            "Cancel":
                -- If the cancel button was not a
                --- #cancelPushButton, you can handle it here.
            "Button Name":
                -- Perform action for button "Button Name"
            otherwise:
                put "Button not implemented"
        end case
    #bitmap:
        -- Check #title to determine bitmap clicked
    #label:
        -- You will probably ignore these events
        -- as labels are not usually clickable
end case

#windowOpening:
-- You will usually ignore #windowOpening events, but
-- you could enable buttons based on movie state
#windowClosed:
-- The dialog will close without your intervention.
-- You will probably want to commit any changes
#windowZoomed :
-- There is no convenient way to zoom a window
-- You'll probably ignore these events, and should
```

**DRAFT, August 3, 1999**

*Lingo in a Nutshell*, published by O'Reilly & Associates  
Copyright ©1996-1999. Bruce A. Epstein. All Rights Reserved.  
Please send feedback to [lingonut@zeusprod.com](mailto:lingonut@zeusprod.com)

```
-- avoid using a zoom box in your dialog's title bar.
#windowResized :
-- There is no convenient way to resize a window, and
-- nothing currently generates #windowResized events.
otherwise:
    put "Unrecognized event: " && event
end case
end dialogActions
```

### Updating the Dialog

Use the *ItemUpdate()* command from within your callback handler to update the dialog's appearance or contents in response to user actions. It takes this form:

```
| ItemUpdate(gMUIObject, itemNumber, singleItemPropList)
```

where *itemNumber* is the number of the item to update, and *singleItemPropList* is a list of new properties for it. The first item should be available from a global variable, and the last two items are received in your callback handler.

*ItemUpdate()* can be used to:

- Enable or disable buttons, text fields, or sliders based on current settings
- Filter editable text fields
- Update a slider
- Change the range of values listed in a pop-up
- Change a bitmap displayed in the dialog

You typically update an item's *#enabled*, *#value* or *#attributes*, but you can update any property except an item's *#type*. Set the *#height*, *#width*, *#locH*, and *#locV* properties to -1 to leave them the item's size and position unchanged.

The *widgeProps* passed to the callback handler only contains the properties for the widget receiving the event. It does *not* contain the properties of all the widgets, nor is there any way to obtain these from the dialog. This is only a problem if you want to update other widgets based on the one generating the event. For example, you can use the *#value* or *#title* property from the *itemPropList* to detect which button, slider or text field was involved. The *widgeNum* (also received by the callback handler) is used primarily to update the same item using *ItemUpdate()*, not to determine which item received the event.

To modify one or more widgets based on events received by *other* widgets, you must save their properties (used in the *#windowItemList* to initialize the dialog) in a global variable.

To maintain its correct state, you must then *update* that global list within your callback handler each time a relevant event occurs. Only then will you have the needed properties available to update other widgets using *ItemUpdate()*. If you store the entire *#windowItemList*, you can use the *widgeNum* received in the callback handler as the index into that list. All widgets, including dividers and group widgets are included in the widget count.

When using dynamic layout mode, there is no way to anticipate what the

eventual size and position of a given widget will be. To update a widget without changing its size and position, set its `#locH`, `#locV`, `#height` and `#width` properties to -1, which will maintain the previous values.

Whew! Now wasn't *that* special!

## Possible Conflicts and Bugs

### Crash when using #RetryCancel

Specifying the `#RetryCancel` option for the `#buttons` property when creating a MUI Alert dialog may crash in D7.0. I am not sure about its reliability in D7.0.2, or D6.5 and prior. The online examples include a substitute handler to manually construct a Retry/Cancel dialog box.

### Palette Conflicts

Like all Windows system dialog boxes, MUI dialog boxes may appear incorrectly when using 256 colors, unless using the Windows System palette or a custom palette with the first and last ten colors reserved. If using a custom bitmap in your MUI dialog box, make sure the bitmap is mapped to the Windows System palette.

### MIAW Conflicts

The MUI Xtra may also conflict with open MIAWs. Close any MIAWs before using the MUI Xtra, or any feature of Director that uses MUI, such as the Behavior parameters dialog box. See the following technote for additional details:

[http://www.macromedia.com/support/director/ts/documents/unexpected\\_error.htm](http://www.macromedia.com/support/director/ts/documents/unexpected_error.htm)

Myron Mandell reports some success with using the MUI Xtra from MIAWs, but also reports that a modal MUI Alert dialog box cannot be dismissed when called from a modal MIAW on the Macintosh (at least this was the case in D6). He suggests the workaround shown below:

```
if the platform contains "Macintosh" then
    set the modal of the activeWindow to FALSE
    displayMUIalert() -- This is a custom handler
    set the modal of the activeWindow to TRUE
else
    displayMUIalert() -- This is a custom handler
end if
```

### Conflicts on 68K Macs and older versions of the Mac OS

One user reported an error with MUI using Mac OS 7.1.2. The error message was:

```
| Script Error: XTRA not found "MUI"
```

I've experienced conflicts on Mac 68K systems, especially ones running older versions of the OS. MUI may require a recent version of the Appearance Manager Extension. It may also fail to load in low memory. Allocate more memory to your Macintosh projector if in doubt.