Improve this documentation (https://github.com/livecode/livecode-ide/blob/develop-9.5/Documentation/guides/Programming%20a%20User%2

Programming a User Interface

Introduction

The user interface for your application is often one of its most important features. The *LiveCode Script* guide shows how to build a user interface using LiveCode's tools and development environment. In this guide we look at how you can edit, or even build a user interface programmatically. Everything you can do using the built-in tools you can also do programmatically. You can even create and modify a user interface at run time in a standalone application, or provide interactive methods for your users to modify specific aspects of your application. This set of

capabilities allow you to produce applications that construct their interface using XML files or a custom data structure, programmatically construct aspects of complex interfaces, modify their look using user specified parameters, create themed or skinned interface options, build your own interface editing tools that plug-in to LiveCode's IDE and much more. You can also create custom objects and attach your own virtual behaviors and custom properties to them. We recommend you spend a little time becoming familiar with building an interface using the tools in the development environment before creating or editing an interface programmatically.

Referring to Objects

In general, you can refer to any object by its **name**, **number**, or **ID** property.

Referring to objects by name

You can refer to an object using its object type followed by its name. For example, to refer to a button named "OK", use the phrase `button "OK"`:

```
set the loc of
button "OK" to
32,104
```

To change an object's name, enter a name in the object's property inspector, or use the **set** command to change the object's **name** property:

```
set the name of
field "Old Name" to
"New Name"
select after text
of field "New Name"
```

Referring to objects by number

A control's number is its layer on the card, from back to front. A card's number is its position in the stack. A stack's number is the order of its creation in the stack file. A main stack's number is always zero.

You can refer to an object using its object type followed

by its number. For example, to refer to the third-from-the-back field on a card, use the phrase "field 3":

```
set the backgroundColor of field 3 to blue
```

To change the number of a card or control, change the Layer box in the Size & Position pane of the object's property inspector, or use the **set** command to change the object's **layer** property:

```
set the layer of field "Backmost" to 1
```

> **Tip:** New objects are always created at the top layer. To refer to an object you've just created, use the ordinal **last**:

```
create button
set the name of last button to "My New Button"
```

Referring to objects by ID

Each object in LiveCode has an ID number. The **ID** property never changes (except for stack IDs), and is guaranteed unique within the stack: no two objects in the same stack can have the same **ID** property.

You can refer to an object using its object type, then keyword `ID`, followed by its `ID` number. For example, to refer to a card whose

```
ID
```

property is 1154, use the phrase card `ID` 1154:

```
go to card ID 1154
```

You cannot change an object's **ID** property (except for a stack).

> **Important:** Wherever possible, you should name your objects and refer to them by name instead of using number or ID. Both the number and ID properties will change if objects are copied and pasted. Additionally, your scripts will rapidly become difficult to read if there are many ID or numerical references to objects.

Referring to objects by ordinal

You can refer to an object using its object type followed by the ordinal numbers **first** through **tenth**, or the special ordinals **middle** and **last**. To refer to a random object, use the special ordinal **any**. For example, to refer to the last card in the current stack, use the special ordinal **last**:

```
go to last card
```

The special descriptor ‘this’

Use the **this** keyword to indicate the current stack, or the current card of a stack:

```
set the backgroundColor of this stack to white
send "mouseUp" to this card
set the textFont of this card of stack "Menubar" to "Sans"
```

Control references

A control is any object that can appear on a card. Fields, buttons, scrollbars, images, graphics, players, widgets, and groups are all controls. Stacks, cards, audio clips, and video clips are *not* controls.

You can refer to an object of any of these object types using the word "control", followed by an ID, name, or number:

```
hide control ID 2566
send mouseDown to control "My Button"
set the hilite of control 20 to false
```

If you use a name, as in the expression `control "Thing"`, the reference is to the first control (with the lowest **layer**) that has that name.

When you refer to a control by number using its object type, the reference is to the Nth control of that type. For example, the phrase field 1 refers to the lowest field on the card. This may not be the lowest control, because there may be controls of other types underneath field 1. However, when you refer to a control by number using the word control, the reference is to the Nth control of any type. The phrase control 1 refers to the lowest control on the card, which may be of any type.

> **Tip:** To refer to the object underneath the mouse pointer, use the **mouseControl** function.

Nested Object References

To refer to an object that belongs to another object, nest the references in the same order as the object hierarchy. For example, if there is a button called "My Button" on a card called "My Card", you can refer to the button like this:

```
show button "My Button" of card "My Card"
```

You can mix names, numbers, ordinal references, and IDs in a nested object reference, and you can nest references to whatever depth is required to specify the object. The only requirement is that the order of references be the same as the order of the object hierarchy, going from an object to the object that owns it. Here are some examples:

```
field ID 34 of card "Holder"
player 2 of group "Main" of card ID 20 of stack "Demo"
first card of this stack
```

stack "Dialog" of stack "Main" -- "*Dialog*" *is a substack*

If you don't specify a card in referring to an object that is contained by a card, LiveCode assumes the object is on the current card. If you don't specify a stack, LiveCode assumes the object is in the current stack. You can reference a control in another stack by either of the following methods:

Use a nested reference that includes the name of the stack:

```
field 1 of stack
"My Stack"
graphic "Outline"
`of` card "Tools"
`of` stack "Some
Stack"
```

Set the **defaultStack** property to the stack you want to refer to first. The **defaultStack** specifies the current stack, so you can refer to any object in the **defaultStack** without including a stack name. This example sets a checkbox in the current stack to have the

same setting as a checkbox in another stack called "Other Stack":

```
put the defaultStack into savedDefault -- so you can set it back later
set the defaultStack to "Other Stack"

put the hilite of button "Me" into meSettin -- this button is in "Other Stack"
set the defaultStack to savedDefault

set the hilite of button "Me Too" to meSetting -- this button is in the original stack
```

If an object is in a group, you can include or omit a reference to the group in a nested reference to the object. For example, suppose the current card contains a button called "Guido", which is part of a group called "Stereotypes". You can refer to the button with any of the following expressions:

```
button "Guido"
button "Guido" of
card 5
button "Guido" of
group "Stereotypes"
button "Guido" of
group "Stereotypes"
of card 5
```

If there is no other button named "Guido" on the card, these examples are equivalent. If there is another button with the same name in another group (or on the card, but not in any group), you must either specify the group (as in the second and third examples) or refer to the button by its **ID** property, to be sure you're referring to the correct button.

Avoid using numbers as object names

It is dangerous to set the **name** property of an object to a number.

For example, create three fields on a new card, naming the first one "3", the second one "2", and the third one "1". In other words, the creation order of those three fields is the reverse of the

numerical names you've given them. Now use the message box to:

```
put "This is field 3" into field 3
```

The third field created, the one named "1", has its text set. The number of the field, *not* the **name** set by you, was used by LiveCode as the object reference.

This shows the perils of using a number as the **name** of an object. There are built-in properties of controls (the **layer** and the **number**) that are based on integers, and to use integers in yet another way, in the **name** property, will likely cause unexpected behaviour at best.

This applies to cards as well. The card order supersedes any numerical card **name**.

You can, however, use a name which includes a number safely. For example, you could call you fields "field1", "field2" and "field3".

Now there is no conflict when you use the message box to:

```
put "This is field
3" into field
"field3"
```

You could also construct numerical field names:

```
on mouseUp
   repeat with y =
1 to 3
      put y into
field ("field" & y)
   end repeat
end mouseUp
```

This will put the values "1", "2" and "3" into the fields named "field1", "field2" and "field3". The "field", concatenated with the number, removes the conflict with the control number.

Properties

A property is an attribute of a LiveCode object. Each type of object has many built-in properties, which affect the object's appearance or behavior. You can also define custom properties for any object, and use them to store any kind of data.

This topic discusses how to use properties, how properties are inherited between objects, and how to create and switch between collections of property settings.

To fully understand this topic, you should know how to create objects, how to use an object's property inspector, and how to write short scripts.

Using Object Properties

A property is an attribute of an object, and each object type has its own set of built-in properties appropriate for that type. An object can be *completely described* by its built-in properties; if you could make all the properties of two objects identical, they'd be the same object. It is thus possible to describe an object entirely as an array of properties, or to export and import properties using text or XML files. More details on some of the methods you can use to do this are covered later in this chapter.

> **Note:** Since no two objects can have the same **ID** property, it's not possible in practice for two different objects to become the same object, because the **ID** will always be different.

Built-in properties determine the appearance and behavior of stacks and their contents - fonts, colors, window types, size and placement, and much more – as well as much of the behavior of the LiveCode application. By changing properties, you can change almost any aspect of your application. When you combine the ability to change properties with the ability to create and delete objects programmatically, you can modify every aspect of your application (subject to any limitations of the license agreement applicable to your edition of LiveCode).

Referring to properties

Property references consist of `the` word `the`, `the` property name, `the` word

of, and a reference to `the` object:

```
the armedIcon of
button "My Button"
the borderWidth of
field ID 2394
the name of card 1
of stack "My Stack"
```

Properties are sources of value, so you can get the value of a property by using it in an expression:

```
put the height of
field "Text" into
myVar
put the width of
image "My Image" +
17 after field
"Values"
if item 1 of the
location of me >
zero then beep
```

For example, to use the **width** property of a button as part of an arithmetic expression, use a statement like the following:

```
add the width of
button "Cancel" to
totalWidths
```

The value of the property – in this case, the width of the button in pixels – is substituted for the property reference when the statement is executed.

To see a list of all the language words (including properties) applicable to a particular object type, open the *Dictionary* and navigate to the entry for the object type. You can filter the entries to only show object types by clicking on the *object* in the type filter section.

Changing properties

To change the value of a property, you use the **set** command:

```
set the borderColor
of group "My Group"
to "red"
set the top of
image ID 3461 to
zero
```

You can also see and change many of an object's properties by selecting the object and choosing *Object Inspector*. See the chapter *Building a User Interface* for more details.

Most built-in properties affect the appearance or behavior of the object. For example, a button's **height**, **width**, and **location** are properties of the button. Changing these properties in a handler causes the button's appearance to change. Conversely, dragging or resizing the button changes the related properties.

Read-only properties

Some properties can be read, but not set. These are called read-only properties. Trying to set a read-only property causes an execution error.

To find out whether a property is read-only, check its entry in the LiveCode Dictionary.

Changing a part of a property

Properties are not containers, so you cannot use a chunk expression to change a *part* of the property. However, you can use a chunk expression to *examine* part of a property. For example, you cannot set line 1 of a property to a new value: you must set the whole property. For more

details see the section *Chunk Expressions* in the chapter on *Processing Text and Data*.

To change one part of a property, first put the property value into a variable, change the required part of the variable, then set the property back to the new variable contents:

```
put the rect of me
into tempRect
put "10" into item
2 of tempRect
set the rect of me
to tempRect
```

Custom properties and virtual properties

A custom property is a property that you define. You can create as many custom properties for an object as you want, and put any kind of data into them, including binary data or array data. You can even store a file in a custom property.

Virtual properties are custom properties that trigger a custom script action when you change them, allowing you to implement "virtual" object behaviors.

Custom properties and *virtual properties* are covered in their respective sections later in this chapter.

Property Inheritance

Most properties are specific to the object they are part of, and affect only that object.

However, some properties of an object, such as its color and text font, take on the settings of the object above it in the object hierarchy. For example, if a field's background color property is not specified (that is, if its **backgroundColor** property is empty), the field takes on the background color of the card that owns it. If no background color is specified for the card either, the stack's background color is used, and so on. This means you can set a background color for a stack, and every object in it will automatically use that background color, without your having to set it for each object.

This process of first checking the object, then the object's owner, then the object that owns that object, and so on, is called *inheritance* of

properties. Each object inherits the background color of the object above it in the hierarchy. Similar inheritance rules apply to the **foregroundColor**, **topColor**, **bottomColor**, **borderColor**, **shadowColor**, and **focusColor** properties, to their corresponding pattern properties, and to the **textFont**, **textSize**, and **textStyle** properties.

Overriding inheritance

Inheritance is used to determine an object's appearance only if the object itself has no setting for the property. If an inheritable property of an object is not empty, that setting overrides any setting the object might inherit from an object above it in the object hierarchy.

For example, if a button's **backgroundColor** property is set to a color reference instead of being empty, the button uses that background color, regardless of the button's owners. If the object has a color of its own, that color is always used.

The effective keyword

If an inheritable property of an object is empty, you can't simply check the property to find out what color or font settings the object displays. In this case, use the **effective** keyword to obtain the inherited setting of the property. The **effective** keyword searches the object's owners, if necessary, to find out what setting is *actually* used.

For example, suppose you have a field whose **textFont** property is empty. The **textFont** of the card that the field is on is set to "Helvetica", so the field inherits this setting and displays its text in the Helvetica font. To find out what font the field is using, use the expression `the effective textFont`:

```
get the textFont of field "My Field" *-- empty*
get the effective textFont of field "My Field" *-- Helvetica*
```

You can use the **effective** keyword with any inherited property.

Global Properties

LiveCode also has global properties, which affect the *overall behavior* of the application. Global properties are accessed and changed the same way as object properties. They do not belong to any particular object, but otherwise they behave like object properties.

> **Tip:** To see a list of all global properties, open the *Message Box*, and choose the *Global Properties* icon – the third icon from the left at the top of the window. To see a list of all properties in the language, including both global and object properties, use the propertyNames global property.

A few properties are *both* global *and* object properties. For example, the **paintCompression** is a global property, and also a property of images. For these

properties, the global setting is separate from the setting for an individual object.

Some other global properties are affected by *system settings*. For example, the default value of the **playLoudness** property is set by the operating system's sound volume setting.

Referring to global properties

You refer to global properties using **the** and the property name:

```
the defaultFolder
the
emacsKeyBindings
the fileType
```

Since global properties apply to the whole application, you don't include an object reference when referring to them.

Global properties are sources of value, so you can get the value of a global property by using it in an expression:

```
get the stacksInUse
put the recentNames
into field "Recent
Cards"
if the ftpProxy is
empty then exit
setMyProxy
```

Changing global properties

To change a global property, you use the **set** command, in the same way as for object properties:

```
set the
itemDelimiter to
"/"
set the grid to
false
set the idleTicks
to 10
```

Some global properties can be changed by other commands. For example, the **lockScreen** property can either be set directly, or changed using the **lock screen** and

```
unlock screen
```

commands. The following two statements are equivalent:

```
set the lockScreen
to false *-- does
the same thing
as...*
unlock screen
```

Saving and restoring global properties

Object properties are part of an object, so they are saved when the stack containing their object is saved. Global properties, however, are not associated with any object, so they are not saved with a stack. If you change the value of a global property, the change is lost when you quit the application.

If you want to use the same setting of a global property during a different session of your application, you must save the setting – in a Preferences file, in a custom property, or elsewhere in a saved file – and restore it when your application starts up.

Text Related Properties

Normally, properties are applied only to objects or, in the case of global properties, to the entire application. However, a few properties also apply to chunks in a field or to single characters in a field.

Text style properties

Certain text-related properties can be applied either to an entire field or to a chunk of a field:

```
set the textFont of
word 3 of field "My
Field" to "Courier"
set the
foregroundColor of
line 1 of field 2
to "green"
if the textStyle of
the clickChunk is
"bold" then beep
```

The following field properties can be applied to either an entire field or to a chunk of the field:

textFont, textStyle, and textSize

textShift

backgroundColor and foregroundColor

backgroundPattern and foregroundPattern (Unix systems)

Each chunk of a field inherits these properties from the field, in the same way that fields inherit from their owners. For example, if a word's **textFont** property is empty, the word is displayed in the field's font. But if you set the word's **textFont** to another font name, that word – and only that word – is displayed in its own font.

To find the text style of a chunk in a field, whether that chunk uses its own styles or inherits them from the field, use the **effective** keyword:

```
get the effective
textFont of word 3
of field ID 2355
answer the
effective
backgroundColor of
char 2 to 7 of
field "My Field"
```

> **Tip:** If a chunk expression includes more than one style, the corresponding property for that chunk reports "mixed". For example, if the first line of a field has a **textSize** of "12", and the second line has a **textSize** of "24", an expression like the textSize of line 1 to 2 of field "My Field" reports "mixed".

Formatted text properties

The **htmlText**, **RTFText**, and **unicodeText** properties of a chunk are equal to the text of that chunk, along with the formatting information that's appropriate for the property.

For example, if a field contains the text "This is a test.", and the word "is" is boldfaced, the htmlText of word 2 reports "is".

For more information on these properties see the chapter on *Processing Text and Data*, as well as the

individual entries for these properties in the *LiveCode Dictionary*.

The formattedRect and related properties

The **formattedRect** property (along with the **formattedWidth**, **formattedHeight**, **formattedLeft**, and **formattedTop**) reports the position of a chunk of text in a field. These properties are read-only.

The **formattedRect**, **formattedLeft**, and **formattedTop** properties can be used for a chunk of a field, but not the entire field. The **formattedWidth** and **formattedHeight** apply to both fields and chunks of text in a field.

The imageSource, linkText, and visited properties

The **imageSource** of a character specifies an image to be substituted for that character when the field is displayed. You use the **imageSource** to display images inside fields:

```
set the imageSource
of char 17 of field
1 to 49232
set the imageSource
of char thisChar of
field "My Field" to
"http://www.example
.com/banner.jpg"
```

The **linkText** property of a chunk lets you associate hidden text with part of a field's text. You can use the **linkText** in a **linkClicked** handler to specify the destination of a hyperlink, or for any other purpose.

The **visited** property specifies whether you have clicked on a text group during the current session. You can get the visited property for any chunk in a field, but it is meaningless unless the chunk's **textStyle** includes "link".

The **imageSource**, **linkText**, and **visited** properties are the only properties that can be set to a chunk of a field, but not to the entire field or any other object. Because they are applied to text in fields, they are listed as field properties in the LiveCode Dictionary.

Creating and Deleting Objects

LiveCode allows you to create and delete objects programmatically. You may optionally specify all the properties for a new object before creating it.

The Create Object Command

You use the **create** command to create a new object.

```
create [invisible]
*type* [*name*] [in
*group*]
```

The *type* is any control that can be on a card: field, button, image, scrollbar, graphic, or player.

The *name* is the name of the newly created object. If you don't specify a *name*, the object is created with a default name.

The *group* is any group that's on the current card. If you specify a *group*, the new object is a member of the group, and exists on each card that has the group. If

you don't specify a group, the object is created on the current card and appears only on that card.

```
create button
"Click Me"
create invisible
field in first
group
```

You can also create widgets of a given kind on a card, using the **create widget** command.

```
create [invisible]
widget [*name*] as
*kind* [in *group*]
```

The *kind* is the kind of any currently installed widget.

For more details, see the *create command* in the *LiveCode Dictionary*. For details on how to specify the properties of an object before creating it, see the section on *Creating Objects Off-screen Using Template Objects*, below.

The Delete Object Command

You can use the **delete** command to remove objects from the stack.

delete {*object}*

The *object* is any available object.

```
delete this card
delete button "New Button"
```

For more details, see the *delete command* in the *LiveCode Dictionary*.

Creating Objects Off-screen Using Template Objects

LiveCode uses *template objects* to allow you to specify the properties for an object before it is created. The template objects are off-screen models – there is one for each possible type of object, e.g. button, field, graphic, etc.

If you need to create a new object and then set some properties on the object, it is more efficient to make the changes to the template object, then create the object. Because the object

can be created with all of its properties set correctly, there is no need to lock the screen and update or reposition the object after creating it. For example, the LiveCode development environment uses the template objects internally to create new objects from the main tool palette.

You set properties on template objects in the same way you set properties on normal objects.

set the {property} of the template{Objecttype} to {value}

For example, to create a button with the name "Hello World", positioned at 100,100:

```
set the name of the
templateButton to
"Hello World"
set the location of
the templateButton
to 100,100
create button
```

When you have used the templateObject to create a new object, you should reset it before using it again.

Resetting the templateObject sets all off its properties back to defaults.

reset the template[Objecttype]

For example, to reset the `templateButton`:

```
reset the templateButton
```

For more details on the template objects, search the *LiveCode Dictionary* for "*template*".

Property Arrays using the Properties Property

In addition to retrieving individual object properties, you can retrieve or set an entire set as an array using the **properties** property. You can use this to edit, copy, export or import properties.

```
set the properties of *object* to *propertiesArray*
put the properties of *object* into propertiesArray
```

The **properties** of an object is an array containing that object's significant built-in properties.

```
put the properties
of button 1 into
myArray
set the properties
of last player to
the properties of
player "Example"
```

Tip: This example handler shows you how to write the properties of an object to a text file.

```
on mouseUp
    put the
properties of
button 1 into
tPropertiesArray
    combine
tPropertiesArray
using return and
"|"
    ask file "Save
properties as:"
    if it is not
empty then put
tPropertiesArray
into URL ("file:" &
it)
end mouseUp
```

In this example, each property name will be written followed by the "|" character and the property value and then a return character.

For more details, see the *properties property* in the *LiveCode Dictionary*.

Custom Properties

A custom property is a property that you create for an object, in addition to its built-in properties. You can define custom properties for any object, and use them to store any kind of data.

This topic discusses how to *create* and *use* custom properties, and how to *organize* custom properties into *sets* (or *arrays*). The following section covers how to create virtual properties and use **getProp** and **setProp** handlers to handle custom property requests.

Using Custom Properties

A custom property is a property that you define. You can create as many custom properties for an object as you want, and put any kind of

data into them (even binary data). You can even store a file in a custom property.

Use a custom property when you want to:

- associate data with a specific object
- save the data with the object in the stack file
- access the data quickly

Creating a Custom Property

You create a custom property by setting the new property to a value. If you set a custom property that doesn't exist, LiveCode automatically creates the custom property and sets it to the requested value.

This means that you can create a custom property in a handler or the message box, simply by using the **set** command. The following statement creates a custom property called "endingTime" for a button:

```
set the endingTime
of button "Session"
to the long time
```

You can create custom properties for any object. However, you cannot create global custom properties, or custom properties for a chunk of text in a field. Unlike some built-in properties, a custom property applies only to an object.

> **Important:** Each object can have its own custom properties, and custom properties are not shared between objects. Creating a custom property for one object does not create it for other objects.

The Content of a Custom Property

You set the value of a custom property by using its property name together with the **set** command, in the same way you set built-in properties:

```
set the myCustomProperty of button 1 to false
```

You can see and change all of an object's custom properties in the Custom Properties pane of the object's property inspector: click the custom property you want to change, then enter the new value.

Changing a part of a property

Like *built-in properties*, custom properties are *not* containers, so you cannot use a chunk expression to change a part of the custom property. Instead, you put the property's value into a variable and change the variable, then set the custom property back to the new variable contents:

```
put the lastCall of
this card into
myVar
put "March" into
word 3 of myVar
set the lastCall of
thisCard to myVar
```

Custom Property Names

The name of a custom property must consist of a single word and may contain any combination of letters, digits, and underscores (_).

The first character must be either a letter or an underscore.

Avoid giving a custom property the same name as a variable. If you refer to a custom property in a handler, and there is a variable by the same name, LiveCode uses the contents of the variable as the name of the custom property. This usually causes unexpected results.

> **Important:** It is important to avoid giving custom properties the same name as existing engine properties *unless* those properties are only ever accessed in the context of a custom property set. Unintended effects may result if you attempt to use the name of an engine property as a custom property.

Custom property names beginning with "**rev**" are reserved for LiveCode's own custom properties. Naming a custom property with a

reserved name may produce unexpected results when working in the development environment.

Referring to Custom Properties

Custom property references look just like built-in property references: the word the, the property name, the word of, and a reference to the object.

For example, to use a custom property called "lastCall" that belongs to a card, use a statement like the following:

```
put the lastCall of
this card into
field "Date"
```

Like built-in properties, custom properties are sources of value, so you can get the value of a custom property by using it in an expression. The property's value is substituted for the property reference when the statement is executed. For example, if the card's "lastCall" custom property is "Today", the example

statement above puts the string "Today" into the "Date" field.

Nonexistent Custom Properties

Custom properties that don't exist evaluate to empty. For example, if the current card doesn't have a custom property called "astCall", the following statement empties the field:

put the lastCall of this card into field "Date" – *empty*

> **Note:** Referring to a nonexistent custom property does not cause a script error. This means that if you misspell a custom property name in a handler, you won't get an error message, so you might not notice the problem right away.

Finding out Whether a Custom Property Exists

The **customKeys** property of an object lists the object's custom properties, one per

line:

```
put the customKeys
of button 1 into
field "Custom
Props"
```

To find out whether a custom property for an object exists, you check whether it's listed in the object's **customKeys**. The following statement checks whether a player has a custom property called "doTellAll":

```
if "doTellAll" is
among the lines of
the customKeys of
player "My Player"
then*...*
```

You can also look in the *Custom Properties* pane of the object's property inspector, which lists the custom properties. See the chapter on *Building a User Interface* for more details.

Custom Properties & Converting Text Between Platforms

When you move a stack developed on a Mac OS or OS X system to a Windows

or Unix system (or vice versa), LiveCode automatically translates text in fields and scripts into the appropriate character set. However, text in custom properties is not converted between the ISO and Macintosh character sets. This is because custom properties can contain binary data as well as text, and converting them would garble the data.

Characters whose ASCII value is between 128 and 255, such as curved quotes and accented characters, do not have the same ASCII value in the Mac OS character set and the ISO 8859-1 character set used on Unix and Windows systems. If such a character is in a field, it is automatically translated, but it's not translated if it's in a custom property.

Because of this, if your stack displays custom properties to the user--for example, if the stack puts a custom property into a field--and if the text contains special characters, it may be displayed incorrectly if you move the

stack between platforms. To avoid this problem, use one of these methods:

Before displaying the custom property, convert it to the appropriate character set using the **macToISO** or **ISOToMac** function. The following example shows how to convert a custom property that was created on a Mac OS system, when the property is displayed on a Unix or Windows system:

```
if the platform is
"MacOS" then
    answer the
myPrompt of button
1
else
    answer
macToISO(the
myPrompt of button
1)
end if
```

Instead of storing the custom property as text, store it as HTML, using the **HTMLText** property of fields:

```
set the myProp of
this card to the
HTMLText of field 1
```

Because the **HTMLText** property encodes special characters as entities, it ensures that the custom property does not contain any special characters--only the platform-independent encodings for them. You can then set a field's **HTMLText** to the contents of the custom property to display it:

```
set the HTMLText of
field "Display" to
the myProp of this
card
```

Storing a file in a custom property

You can use a URL to store a file's content in a custom property:

```
set the
myStoredFile of
stack "My Stack" to
URL
"binfile:mypicture.
jpg"
```

You restore the file by putting the custom property's value into a URL:

```
put the
myStoredFile of
stack "My Stack"
into URL
"binfile:mypicture.
jpg"
```

Because a custom property can hold any kind of data, you can store either text files or binary files in a custom property. You can use this capability to bundle media files or other files in your stack.

Many Mac OS Classic files have a resource fork. To store and restore such a file, you can use the **resfile** URL scheme to store the content of the resource fork separately.

> **Tip:** To save space, compress the file before storing it:

```
set the
myStoredFile of
stack "My Stack" to
compress(URL
"binfile:mypicture.
jpg")
```

When restoring the file, decompress it first:

```
put decompress(the
myStoredFile of
stack "My Stack")
into URL
"binfile:mypicture.
jpg"
```

For more information about using URL containers, see the section on *Working with Files, URLs and Sockets* in the *Transferring Information* guide.

Deleting a custom property

As described above, the **customKeys** property of an object is a list of the object's custom properties. You can set the **customKeys** of an object to control which custom properties it has.

In LiveCode, there is no command to delete a custom property. Instead, you place all the custom property names in a variable, delete the one you don't want from that variable, and set the object's **customKeys** back to the modified contents of

the variable. This removes the custom property whose name you deleted.

For example, the following statements delete a custom property called "propertyToRemove" from the button "My Button":

```
get the customKeys
of button "My
Button"
set the
wholeMatches to
true
delete line
lineOffset("propert
yToRemove",it) of
it
set the customKeys
of button "My
Button" to it
```

You can also delete a custom property in the *Custom Properties* pane of the object's *Property Inspector*. Select the property's name and click the *Delete* button to remove it.

Custom Property Sets

Custom properties can be organized into *custom property sets* – or arrays of custom properties. A custom

property set is a group of custom properties that has a name you specify.

When you refer to a custom property, LiveCode looks for that property in the object's currently-active custom property set. When you create or set a custom property, LiveCode creates it in the currently-active custom property set, or sets the value of that property in the currently-active set. One custom property set is active at any one time, but you can use array notation to get or set custom properties in sets other than the current set.

The examples in the previous section assume that you haven't created any custom property sets. If you create a custom property without creating a custom property set for it, as shown in the previous examples, the new custom property becomes part of the object's default custom property set.

Creating custom property sets

To make a custom property set active, you set the object's **customPropertySet**

property to the set you want to use. As with custom properties and local variables, if the custom property set you specify doesn't exist, LiveCode automatically creates it, so you can create a custom property set for an object by simply switching to that set.

The following statement creates a custom property set called "Alternate" for an object, and makes it the active set:

```
set the customPropertySet of the target to "Alternate"
```

The statement above creates the custom property set.

You can also view, create, and delete custom property sets in the Custom pane of the object's property inspector.

You can list all the custom property sets of an object using its **customPropertySets** property.

As with custom properties, you can create custom property sets for any object. But you can't create global custom property sets, or custom property sets for a chunk of a field.

Custom property set names

The names of custom property sets should consist of a single word, with any combination of letters, digits, and underscores (_). The first character should be either a letter or an underscore.

It is possible to create a custom property set with a name that has more than one word, or that otherwise doesn't conform to these guidelines. However, this is not recommended, because such a custom property set can't be used with the array notation described below.

> **Note:** When you use the Custom Properties pane in the property inspector to create a custom property set, the pane restricts you to these guidelines.

Referring to custom property sets

To switch the active custom property set, set the object's **customPropertySet** property to the name of the set you want to use:

```
set the customPropertySet of button 3 to "Spanish"
```

Any references to custom property refer to the current custom property set. For example, suppose you have two custom property sets named "Spanish" and "French", and the French set includes a custom property called "Paris" while the Spanish set does not. If you switch to the Spanish set, the **customKeys** of the object does not include "Paris",

because the current custom property set doesn't include that property.

If you refer to a custom property that isn't in the current set, the reference evaluates to empty. If you set a custom property that isn't in the current set, the custom property is created in the set. You can have two custom properties with the same name in different custom property sets, and they don't affect each other: changing one does not change the other.

The **customProperties** property of an object includes only the custom properties that are in the current custom property set. To specify the **customProperties** of a particular custom property set, you include the set's name in square brackets:

```
put the customProperties[mySet] of this card into myArray
```

Finding out whether a custom property set exists

The **customPropertySets** property of an object lists the object's custom property sets, one per line:

```
answer the
customPropertySets
of field "My Field"
```

To find out whether a custom property set for an object exists, you check whether it's listed in the object's **customPropertySets**. The following statement checks whether an image has a custom property set called "Spanish":

```
if "Spanish" is
among the lines of
the
customPropertySets
of image ID 23945
then*...*
```

You can also look in the Custom Properties pane of the object's property inspector, which lists the custom property sets in the "Set" menu halfway down the pane.

The default custom property set

An object's default custom property set is the set that's active if you haven't used the **customPropertySet** property to switch to another set. Every object has a default custom property set; you don't need to create it.

If you create a custom property without first switching to a custom property set – as in the earlier examples in this topic – the custom property is created in the default set. If you don't set the **customPropertySet** property, all your custom properties are created in the default set.

The default custom property set has no name of its own, and is not listed in the object's **customPropertySets** property. To switch from another set to the default set, you set the object's **customPropertySet** to empty:

```
set the
customPropertySet
of the target to
empty
```

Using multiple custom property sets

Since only one custom property set can be active at a time, you can create separate custom properties with the same name but different values in different sets. Which value you get depends on which custom property set is currently active.

A translation example

Suppose your stack uses several custom properties that hold strings in English, to be displayed to the user by various commands. Your stack might contain a statement such as this:

answer the standardErrorPrompt of this stack

The statement above displays the contents of the custom property called "standardErrorPrompt" in a dialog box.

Suppose you decide you want to translate your application into French. To do this, you make your original set of English custom properties into a custom property set (which you might call "myEnglishStrings"), and create a new set called "myFrenchStrings" to hold the translated properties.

Each set has the same-named properties, but the values in one set are in French and the other in English. You switch between the sets depending on what language the user chooses. The statement:

answer the standardErrorPrompt of this stack

provides either the English or French, depending on which custom property set is active: "myEnglishStrings" or "myFrenchStrings".

Copying custom properties between property sets

When it's created, a custom property set is empty, that is, there aren't any custom

properties in it. You put custom properties into a new custom property set by creating the custom properties while the set is active:

```
-- create new set and make it active:
set the customPropertySet of button 1 to "MyNewSet"
-- now create a new custom property in the current set:
set the myCustomProp of button 1 to true
```

You can also use the **customProperties** property (which was discussed earlier in this topic) to copy custom properties between sets. For example, suppose you have created a full set of custom properties in a custom property set called "myEnglishStrings", and you want to copy them to a new custom property set, "frenchStrings", so you can translate them easily. The following statements create the new custom property set,

then copy all the properties from the old set to the new one:

```
-- create the new set:
set the customPropertySet of this stack to "frenchStrings"
-- copy the properties in the English set to the new set:

set the customProperties["frenchStrings "] of this stack to the customProperties["frenchStrings "] of this stack
```

> **Caution:** Custom property sets in the development environment

Arrays, custom properties, and custom property sets

All the custom properties in a custom property set form an array. The array's name is the custom property set name, and the elements of the array are the individual custom properties in that custom property set.

Referring to custom properties using array notation

You can use array notation to refer to custom properties in any custom property set. This lets you get and set any custom property, even if it's not in the current set, without changing the current set.

For example, suppose a button has a custom property named "myProp" which is in a custom property set called "mySet". If "mySet" is the current set, you can refer to the "myProp" property like this:

```
get the myProp of
button 1
set the myProp of
the target to 20
```

But you can also use array notation to refer to the "myProp" property, even if "mySet" is not the current set. To refer to this custom property regardless of which custom property set is active, use statements like the following:

```
get the
mySet["myProp"] of
button 1
set the
mySet["myProp"] of
the target to 20
```

> **Note:** Because the default custom property set has no name, you cannot use array notation to refer to a custom property in the default set.

Storing an array in a custom property set

If you store a set of custom properties in a custom property set, the set can be used just like an array. You can think of the custom property set as though it were a single custom property, and the properties in the set as the individual elements of the array.

To store an array variable as a custom property set, use a statement like the following:

```
set the
customProperties["m
yProperty"] of me
to theArray
```

The statement above creates a custom property set called "myProperty", and stores each element in "theArray" as a custom property in the new set. To retrieve a single element of the array, use a statement like this:

```
get the
myProperty["myEleme
nt"] of field
"Example"
```

Deleting a custom property set

As described above, the **customPropertySets** property of an object is a list of the object's custom property sets. You can set the **customPropertySets** of an object to control which custom property sets it has.

In LiveCode, there is no command to delete a custom property set. Instead, you place all the custom property set names in a variable,

delete the one you don't want from that variable, and set the **customPropertySets** back to the modified contents of the variable. This removes the custom property set whose name you deleted.

For example, the following statements delete a custom property set called "mySet" from the button "My Button":

```
get the customPropertySets of button "My Button"
set the wholeMatches to true
delete line lineOffset("mySet", it) of it
set the customPropertySets of button "My Button" to it
```

You can also delete a custom property set in the Custom Properties pane of the object's property inspector. Select the set's name from the Set menu, then click the Delete button to remove it.

Attaching Handlers to Custom Properties

When you change a custom property, LiveCode sends a **setProp** trigger to the object whose property is being changed. You can write a **setProp** handler to trap this trigger and respond to the attempt to change the property. Like a message, this trigger uses the message path, so you can place the **setProp** handler anywhere in the object's message path.

Similarly, when you get the value of a custom property, LiveCode sends a **getProp** call to the object whose property is being queried. You can write a **getProp** handler to reply to the request for information. Like a function call, the **getProp** call also traverses the message path.

Using **getProp** and **setProp** handlers, you can:

- validate a custom property's value before setting it
- report a custom property's value in a format other than what it's stored as
- ensure the integrity of a collection of properties by setting them all at

once
- change an object's behavior when a custom property is changed

setProp triggers and **getProp** calls are not sent when a built-in property is changed or accessed. They apply only to custom properties.

Responding to changing a custom property

When you use the **set** command to change a custom property, LiveCode sends a **setProp** trigger to the object whose property is being changed.

A **setProp** trigger acts very much like a message does. It is sent to a particular object. If that object's script contains a **setProp** handler for the property, the handler is executed; otherwise, the trigger travels along the message path until it finds a handler for the property. If it reaches the end of the message path without being trapped, the **setProp** trigger sets the custom property to its new value. For more

information about the message path, see the section on the *Message Path*.

You can include as many **setProp** handlers in a script for as many different custom properties as you need.

The structure of a *setProp* handler

Unlike a message handler, a `setProp` handler begins with the word `setProp` instead of the word `on`. This is followed by the handler's name (which is the same as the name of the custom property) and a parameter that holds the property's new value. A `setProp` handler, like all handlers, ends with the word "end" followed by the handler's name.

The following example shows a setProp handler for a custom property named "percentUsed", and can be placed in the script of the object whose custom property it is:

```
setProp
percentUsed
newAmount
    -- responds
to setting the
percentUsed
property
    if newAmount is
not a number or
newAmount < zero or
newAmount > 100
then
        beep 2
        exit
percentUsed
    end if
    pass
percentUsed
end percentUsed
```

When you set the "percentUsed" custom property, the "percentUsed" handler is executed:

```
set the percentUsed
of scrollbar
"Progress" to 90
```

When this statement is executed, LiveCode sends a **setProp** trigger to the scrollbar. The new value of 90 is placed in the *newAmount* parameter. The handler makes sure that the new value is in the range 0–100; if not, it beeps and exits the handler, preventing the property from being set.

For more details about the **setProp** control structure, see **setProp** in the LiveCode Dictionary.

Passing the setProp trigger

When the **setProp** trigger reaches the engine - the last stop in the message path - the custom property is set. If the trigger is trapped and doesn't reach the engine, the custom property is not set.

To let a trigger pass further along the message path, use the **pass** control structure. The **pass** control structure stops the current handler and sends the trigger on to the next object in the message path, just as though the object didn't have a handler for the custom property.

In the "percentUsed" handler above, if the *newAmount* is out of range, the handler uses the **exit** control structure to halt; otherwise, it executes the **pass** control structure. If the *newAmount* is in the right range, the pass control structure lets the property be set. Otherwise, since the trigger is not passed, it never reaches the engine, so the property is not changed.

You can use this capability to check the value of any custom property before allowing it to be set. For example, if a custom property is supposed to be boolean (true or false), a setProp handler can trap the trigger if the value is anything but true or false:

```
setProp myBoolean newValue
    if newValue is true or newValue is false
    then pass myBoolean
exit myBoolean
```

Using the message path with a setProp trigger

Because **setProp** triggers use the message path, a single object can receive the **setProp** triggers for all the objects it owns. For example, **setProp** triggers for all controls on a card are sent to the card, if the control's script has no handler for that property. You can take advantage of the message path to implement the same **setProp** behavior for objects that all have the same custom property.

If a **setProp** handler sets its custom property, for an object that has that **setProp** handler in its message path, a runaway recursion will result. To avoid this problem, set the **lockMessages** property to true before setting the custom property.

> **Note:** To refer to the object whose property is being set, use the **target** function. The **target** refers to the object that first received the **setProp** trigger
>
> - the object whose custom property is being set - even if the handler being executed is in the script of another object.

Setting properties within a setProp handler

In the "lastChanged" example in the box above, the handler sets the custom property directly, instead of simply passing the **setProp** trigger. You must use this method if the handler makes a change to the property's value, because the **pass** control structure simply passes on the original value of the property.

If you use the **set** command within a **setProp** handler to set the same custom property for the current object, no **setProp** trigger is sent to the target object (this is to avoid runaway recursion, where the **setProp** handler triggers itself). Setting a different custom property sends a **setProp** trigger. So does setting the handler's custom property for an object other than the one whose script contains the **setProp** handler.

Using this method, you can not only check the value of a property, and allow it to be set only if it's in range, you can also change the value so that it is in the correct range, has the correct format, and so on.

The following example is similar to the "percentUsed" handler above, but instead of beeping if the *newAmount* is out of range, it forces the new value into the range 0–100:

```
setProp
percentUsed
newAmount
    set the
percentUsed of the
target to
max(zero,min(100,ne
wAmount))
end percentUsed
```

Nonexistent properties

If the custom property specified by a setProp handler doesn't exist, the setProp handler is still executed when a handler sets the property. If the handler passes the **setProp** trigger, the custom property is created.

Custom property sets and setProp handlers

A setProp handler for a custom property set behaves differently from a setProp handler for a custom property that's in the default set.

When you set a custom property in a custom property set, the **setProp** trigger is named for the set, not the property. The property name is passed in a parameter using a special notation. This means that, for custom properties in a set, you write a single setProp handler for the set, rather than one for each individual property.

The following example handles **setProp** triggers for all custom properties in a custom property set called *myFrenchStrings*, which contains custom properties named *standardErrorPrompt*, *filePrompt*, and perhaps other custom properties:

```
setProp
myFrenchStrings[myP
ropertyName]
newValue
-- The
myPropertyName
parameter
contains the name
of
-- the property
that's being set
    switch
myPropertyName
    case
"standardErrorPromp
t"
        set the
myFrenchStrings["st
andardErrorPrompt"]
of the target to
return & newValue
& return
        exit
myFrenchStrings
    break
    case
"filePrompt"
        set the
myFrenchStrings["fi
lePrompt"] of the
target to return&
newValue & return
    exit
myFrenchStrings
    break
    default
        pass
myFrenchStrings
    end switch
end myFrenchStrings
```

As you can see from the **exit**, **pass**, and **end** control structures, the name of this setProp handler is the same as the name of the custom property set that it controls

- "myFrenchStrings".

Because there is only one handler for all the custom properties in this set, the handler uses the **switch** control structure to perform a different action for each property that it deals with.

Suppose you change the "standardErrorPrompt" custom property:

```
set the customPropertySet of this stack to "myFrenchStrings"
set the standardErrorPrompt of this stack to field 1
```

LiveCode sends a **setProp** trigger to the stack, which causes the above handler to execute. The property you set – "standardErrorPrompt" – is placed in the "myPropertyName" parameter, and the new value--the contents of field 1 – is placed in the "newValue" parameter. The handler executes the case for "standardErrorPrompt", putting a **return** character before and after the property before setting it.

If you set a custom property other than "standardErrorPrompt" or "filePrompt" in the "myFrenchStrings" set, the default case is executed. In this case, the **pass** control structure lets the **setProp** trigger proceed along the message path, and when it reaches the engine, LiveCode sets the custom property.

> **Note:** As mentioned above, you can address a custom property in a set either by first switching to that set, or using array notation to specify both set and property. The following example:

```
set the
customPropertySet
of me to "mySet"
set the myProperty
of me to true
```

is equivalent to:

```
set the
mySet["myProperty"]
of me to true
```

Regardless of how you set the custom property, if it is a member of a custom property set, the **setProp** trigger has the name of the set - not the custom property itself - and you must use a setProp handler in the form described above to trap the **setProp** trigger.

Responding to a request for the value of a custom property

When you use a custom property in an expression, LiveCode sends a getProp call to the object whose property's value is being requested.

A getProp call acts very much like a custom function call. It is sent to a particular object. If that object's script contains a **getProp** handler for the property, the handler is executed, and LiveCode substitutes the value it returns for the custom property reference. Otherwise, the call travels

along the message path until it finds a handler for the property. If the getProp call reaches the end of the message path without being trapped, LiveCode substitutes the custom property's value in the expression.

You can include as many **getProp** handlers in a script as you need.

The structure of a getProp handler

Unlike a message handler, a `getProp` handler begins with the word `getProp` instead of the word `on`. This is followed by the handler's name (which is the same as the name of the custom property). A `getProp` handler, like all handlers, ends with the word "end" followed by the handler's name.

The following example is a getProp handler for a custom property named "percentUsed":

```
getProp
percentUsed
    global
lastAccessTime
    put the seconds
into lastAccessTime
    pass
percentUsed
end lastChanged
```

When you use the "percentUsed" custom property in an expression, the handler is executed:

```
put the percentUsed
of card 1 into
myVariable
```

When this statement is executed, LiveCode sends a getProp call to the card to retrieve the value of the "percentUsed" property. This executes the getProp handler for the property. The example handler stores the current date and time in a global variable before the property is evaluated. For more details, see *getProp* in the *LiveCode Dictionary*.

Returning a value from a getProp handler

When the getProp trigger reaches the engine – the last stop in the message path – LiveCode gets the custom property from the object and substitutes its value in the expression where the property was used.

To let a trigger pass further along the message path, use the **pass** control structure. The **pass** control structure stops the current handler and sends the trigger on to the next object in the message path, just as though the object didn't have a handler for the custom property.

To report a value other than the value that's stored in the custom property - for example, if you want to reformat the value first – you use the **return** control structure instead of passing the getProp call. The following example is a getProp handler for a custom property named "lastChanged", which holds a date in **seconds**:

```
getProp
lastChanged
get the lastChanged
of the target
convert it to long
date
return it
end lastChanged
```

The **return** control structure, when used in a getProp handler, reports a property value to the handler that requested it. In the above example, the converted date – not the raw property – is what is reported. As you can see from the example, you're not limited to returning the actual, stored value of the custom property. In fact, you can return any value at all from a getProp handler.

> **Important:** If you use a custom property's value within the property's getProp handler, no getProp call is sent to the target object. This is to avoid runaway recursion, where the **getProp** handler calls itself.

A handler can either use the **return** control structure to return a value, or use the **pass** control structure to let LiveCode get the custom property from the object.

If the getProp call is trapped before it reaches the engine and no value is returned in the getProp handler, the custom property reports a value of **empty**. In other words, a getProp handler must include either a **return** control structure or a **pass** control structure, or its custom property will always be reported as empty.

Using the message path with a getProp call

Because getProp calls use the message path, a single object can receive the getProp calls for all the objects it owns. For example, getProp calls for all controls on a card are sent to the card, if the control's script has no handler for that property. You can take advantage of the message path to implement the same **getProp** behavior for objects that all have the same custom property.

If a **getProp** handler is not attached to the object that has the custom property and it uses the value of the custom property, a runaway recursion will result. To avoid this problem, set the **lockMessages** property to true before getting the custom property's value.

Nonexistent properties

If the custom property specified by a getProp handler doesn't exist, the getProp handler is still executed if the property is used in an expression. Nonexistent properties report **empty**; getting the value of a custom property that doesn't exist does not cause a script error.

Custom property sets and getProp handlers

A getProp handler for a custom property set behaves differently from a getProp handler for a custom property that's in the default set.

When you use the value of a custom property in a custom property set, the getProp call is named for the set, not the property. The property name

is passed in a parameter using array notation. This means that, for custom properties in a set, you write a single getProp handler for the set, rather than one for each individual property.

The following example handles getProp calls for all custom properties in a custom property set called *expertSettings*, which contains custom properties named *fileMenuContents*, *editMenuContents*, and perhaps other custom properties:

```
getProp expertSettings[thePropertyName]
    -- The thePropertyName parameter contains the name of
    -- the property that's being set
    switch thePropertyName
    case "fileMenuContents"
        if the expertSettings[fileMenuContents] of the target is empty then return "(No items"
        else pass expertSettings
    break
    case "editMenuContents"
        if the expertSettings[editMenuContents] of the target is empty then return the noviceSettings[editMenuContents] of the target
        else pass expertSettings
    break
    default
        pass expertSettings
    end switch
end expertSettings
```

As you can see from the **pass** and **end** control structures, the name of this getProp handler is the same as the name of the custom property set that it controls –

"expertSettings". Because there is only one handler for all the custom properties in this set, the handler uses the **switch** control structure to perform a different action for each property that it deals with.

Suppose you get the "fileMenuContents" custom property:

```
set the customPropertySet of button 1 to "expertSettings"
put the fileMenuContents of button 1 into me
```

LiveCode sends a getProp call to the button, which causes the above handler to execute. The property you queried – "fileMenuContents" – is placed in the "thePropertyName" parameter. The handler executes the case for "fileMenuContents": if the property is empty, it returns " (No items". Otherwise, the **pass** control structure lets the getProp call proceed along the message path, and

when it reaches the engine, LiveCode gets the custom property.

Virtual Properties

A virtual property is a custom property that exists only in a **setProp** and/or **getProp** handler, and is never actually set. Virtual properties are never attached to the object. Instead, they act to trigger **setProp** or **getProp** handlers that do the actual work.

When you use the **set** command with a virtual property, its setProp handler is executed, but the **setProp** trigger is not passed to the engine, so the property is not attached to the object. When you use a virtual property in an expression, its getProp handler returns a value without referring to the object. In both cases, using the property simply executes a handler.

You can use virtual properties to:

- Give an object a set of behaviors
- Compute a value for an object
- Implement a new

property that acts like a built-in property

When to use virtual properties

Because they're not stored with the object, virtual properties are transient: that is, they are re-computed every time you request them. When a custom property depends on other properties that may be set independently, it's appropriate to use a virtual property.

For example, the following handler computes the current position of a scrollbar as a percentage (instead of an absolute number):

```
getProp asPercentage *-- of a scrollbar*
    put the endValue of the target - the startValue of the target into valueExtent
    return the thumbPosition of me * 100 div valueExtent
end asPercentage
```

The "asPercentage" custom property depends on the scrollbar's **thumbPosition**, which can be changed at any time (either by the user or by a handler). Because of this, if we set a custom property for the object, it would have to be re-computed every time the scrollbar is updated in order to stay current. By using a virtual property, you can ensure that the value of the property is never out of date, because the getProp handler re-computes it every time you call for the "asPercentage" of the scrollbar.

Virtual properties are also useful solutions when a property's value is large. Because the virtual property isn't stored with the object, it doesn't take up disk space, and only takes up memory when it's computed.

Another reason to use a virtual property is to avoid redundancy. The following handler sets the width of an object, not in pixels, but as a percentage of the object's owner's **width**:

```
setProp
percentWidth
newPercentage
    set the width
of the target to
the width of the
owner of the target
* newPercentage div
100
end percentWidth
```

Suppose this handler is placed in the script of a card button in a 320-pixel-wide stack. If you set the button's "percentWidth" to 25, the button's **width** is set to 80, which is 25% of the card's 320-pixel width. It doesn't make much sense to store an object's percentWidth, however, because it's based on the object's **width** and its owner's **width**.

Consider using virtual properties whenever you want to define an attribute of an object, but it doesn't make sense to store the attribute with the object – because it would be redundant, because possible changes to the object mean it would have to be re-computed anyway, or because the property is too large to be easily stored.

Handlers for a virtual property

As you can see by looking at the example above, a handler for a virtual property is structured like a handler for any other custom property. The only structural difference is that, since the handler has already done everything necessary, there's no need to actually attach the custom property to the object or get its value from the object. When you set a virtual property or use its value, the **setProp** trigger or **getProp** call does not reach the engine, but is trapped by a handler first.

Virtual property setProp handlers

A setProp handler for an ordinary custom property includes the **pass** control structure, allowing the **setProp** trigger to reach the engine and set the custom property (or else it includes a **set** command that sets the property directly). A handler for a virtual property, on the other hand, does not include the **pass** control structure, because a virtual property should not be set. Since the

property is set automatically when the trigger reaches the end of the message path, a virtual property's handler does not pass the trigger.

If you examine an object's custom properties after setting a virtual property, you'll find that the custom property hasn't actually been created. This happens because the **setProp** handler traps the call to set the property; unless you pass the setProp trigger, the property isn't passed to LiveCode, and the property isn't set.

Virtual property getProp handlers

Similarly, a getProp handler for an ordinary custom property either gets the property's value directly, or passes the getProp call so that the engine can return the property's value. But in the case of a virtual property, the object doesn't include the property, so the getProp handler must return a value.

Creating new object properties

You can use virtual properties to create a new property that applies to all objects, or to all objects of a particular type. Such a property acts like a built-in property, because you can use it for any object. And because a virtual property doesn't rely on a custom property being stored in the object, you don't need to prepare by creating the property for each new object you create: the virtual property is computed only when you use it in an expression.

The following example describes how to implement a virtual property called "percentWidth" that behaves like a built-in property.

Setting the "percentWidth" property

Suppose you place the "percentWidth" handler described above in a stack script instead of in a button's script:

```
setProp
percentWidth
newPercentage
    set the width
of the target to
the width of the
owner of the target
* newPercentage div
100
end percentWidth
```

Because **setProp** triggers use the message path, if you set the "percentWidth" of any object in the stack, the stack receives the **setProp** trigger (unless it's trapped by another object first). This means that if the handler is in the stack's script, you can set the "percentWidth" property of any object in the stack.

If you place the handler in a backscript, you can set the "percentWidth" of any object, anywhere in the application.

To refer to the object whose property is being set, use the **target** function. The **target** refers to the object that first received the **setProp** trigger--the object whose custom property is being set--even if the handler being executed is in the script of another object.

Getting the "percentWidth" property

The matching getProp handler, which lets you retrieve the "percentWidth" of an object, looks like this:

```
getProp
percentWidth
    return 100 *
(the width of the
target div the
width of the owner
of the target)
end percentWidth
```

If you place the handler above in a card button's script, the following statement reports the button's width as a percentage:

```
put the
percentWidth of
button "My Button"
into field 12
```

For example, if the stack is 320 pixels wide and the button is 50 pixels wide, the button's **width** is 15% of the card **width**, and the statement puts "15" into the field.

Like the **setProp** handler for this property, the **getProp** handler should be placed far along the message path. Putting it in a stack script makes the property available to all objects in the stack; putting it in a backscript makes the property available to all objects in the application.

Limiting the "percentWidth" property

Most built-in properties don't apply to all object types, and you might also want to create a virtual property that only applies to certain types of objects. For example, it's not very useful to get the width of a substack as a percentage of its main stack, or the width of a card as a percentage of the stack's width.

You can limit the property to certain object types by checking the **target** object's name:

```
setProp
percentWidth
newPercentage
    if word 1 of
the name of the
target is "stack"
or word 1 of the
name of the target
is "card" then
exit setProp
    set the width
of the target to
the width of the
owner of the target
* newPercentage div
100
end percentWidth
```

The first word of an object's **name** is the object type, so the above revised handler ignores setting the "percentWidth" if the object is a card or stack.

Property Profiles

A property profile is a collection of object property settings, which is stored as a set. A profile for an object can contain settings for almost any properties of the object.

You can include values for most built-in properties in a profile, and create as many different property profiles as you need for any object. Once you've created a profile, you can switch the

object to the profile to change all the property values that are defined in the profile.

For example, suppose you create a property profile for a field that includes settings for the field's color properties. When you switch to that profile, the field's colors change, while all other properties (not included in the profile) remain the same.

Use property profiles when you want to:

- Create "skins" for your application
- Display your application in different languages
- Present different levels--"novice","expert", and so on
- Use different user-interface standards for different platforms

For details on how to create property profiles using the IDE, see the section on *Property Profiles* in the chapter *Building a User Interface*.

Profile names

Profile names follow the same rules as variable names. A profile name must be a single word, consisting of letters, digits, and underscores, and must start with either a letter or an underscore.

> **Tip:** If you want to use a single command to switch several objects to a particular profile, give the profile the same name for each of the objects it applies to.

The master profile

Every object has a master profile that holds the default settings for all its properties. If you don't set an object's profile, the master profile is used. When you create a new profile, you can change settings for various properties to make them different from the master profile's settings.

If you don't specify a property setting in a profile, the master profile's setting is used, so you don't have to specify all properties of an

object when you create a profile, only the ones you want to change.

By default, the master profile is named "Master". You can change the master profile's name in the *Property Profiles* pane of the *Preferences* window.

Switching between profiles

Switching a single object

To switch an object's profile, you can use either the object's property inspector or the **revProfile** property.

```
set the revProfile
of player "My
Player" to
"MyProfile"
```

Switching all the objects on a card

To switch the profiles of all the objects on a card, use the **revSetCardProfile** command:

```
revSetCardProfile
"MyProfile","My
Stack"
```

The statement above sets the profile of all objects on the current card of the stack named "My Stack". (Although the **revSetCardProfile** command changes a card, you specify a stack name, not a card name.)

If an object on the card does not have a profile with the specified name, the object is left untouched.

Switching all the objects in a stack

To switch the profiles of all the objects in a stack, use the **revSetStackProfile** command:

```
revSetStackProfile "MyProfile","My Stack"
```

The statement above sets the profile of all objects in the stack named "My Stack".

If an object in the stack does not have a profile with the specified name, the object is left untouched.

Switching all the objects in a stack file

To switch the profiles of all the objects in every stack in a stack file, use the **revSetStackFileProfile** command:

```
revSetStackFileProfile "MyProfile","My Stack"
```

The statement above sets the profile of all objects in the stack named "My Stack", along with any other stacks in the same stack file.

If an object in any of the stacks does not have a profile with the specified name, the object is left untouched.

Creating a profile in a handler

In addition to creating property profiles in the property inspector, you can create a profile in a handler.

To enable creating profiles, check the "Create profiles automatically" box in the "Property Profiles" pane of the Preferences window. If this box is checked, setting

the **revProfile** property of an object automatically creates the profile.

This ability is particularly useful if you want to create a number of profiles, as shown in the following example:

```
on mouseUp
    -- creates a profile for each card in the stack
    repeat with thisCard = 1 to the number of cards
        set the revProfile of card x to "myNewProfile"
    end repeat
end mouseUp
```

The handler above creates a profile called "myNewProfile" for all the cards in the current stack.

In order for this handler to work, the "Create profiles automatically" option in the "Property Profiles" pane of the Preferences window must be turned on.

You can control this behavior either in Preferences window or using the **gRevProfileReadOnly** keyword. If you don't want to

save property changes when switching profiles, do one of the following:

Set the **gRevProfileReadOnly** variable to true:

```
global gRevProfileReadOnly
put true into gRevProfileReadOnly
```

In the "Property Profiles" pane of the Preferences window, uncheck the box labeled "Don't save changes in profile".

The two methods of changing this setting are equivalent: changing the **gRevProfileReadOnly** variable also changes the preference setting, and vice versa.

For more details, see *gRevProfileReadOnly* in the *LiveCode Dictionary*.

Adding profile settings in a handler

You can add a property setting to a profile by switching to the profile, then setting the property:

```
set the revProfile
of button 1 to
"MyProfile"
set the
foregroundColor of
button 1 to "red"
set the revProfile
of button 1 to
"Master"
```

By default, if you change a property and then switch profiles, the property you changed and its current setting is saved with the profile.

Managing Windows, Palettes and Dialogs

LiveCode provides complete control over all aspects of window management, including moving, re-layering, and changing window mode.

Moving a window

Usually, you use either the **location** or **rectangle** property of a stack to move the stack window.

The**location** property specifies the center of the stack's window, relative to the top left corner of the main screen. Unlike the

location of controls, the **location** of a stack is specified in absolute coordinates. The following statement moves a stack to the center of the main screen:

```
set the location of stack "Wave" to the screenLoc
```

The **rectangle** property of a stack specifies the position of all four edges, and can be used to resize the window as well as move it:

```
set the rectangle of this stack to "100,100,600,200"
```

> **Tip:** To open a window at a particular place without flickering, set the stack's **location** or **rectangle** property to the desired value either before going to it, or in the stack's preOpenStack handler.

You can also use associated properties to move a window. Changing a stack's **bottom** or **top** property moves the window up or down on the screen. Changing the **left** or **right** property moves the window from side to side.

Changing a window's layer

You bring a window to the front by using the **go** command:

```
go stack "Alpha"
```

If the stack is already open, the **go** command brings it to the front, without changing its mode.

To find out the layer order of open stack windows, use the **openStacks** function. This function lists all open stack windows in order from front to back.

The palette layer

Normally, palette windows float above editable windows and modeless dialog boxes. A palette will always above a standard window, even if you bring the standard window to

the front. This helps ensure that palettes, which usually contain tools that can be used in any window, cannot disappear behind document windows. It's also a good reason to make sure you design palette windows to be small, because other windows cannot be moved in front of them if the palette blocks part of the window.

The system palette layer

System windows--stacks whose **systemWindow** property is true--float above all other windows, in every running application. This means that even if the user brings another application to the front, your application's system windows remain in front of all windows.

System windows are always in front of other windows, and you cannot change this behavior.

The active window

In most applications, commands are applied to the active window. Since LiveCode gives you the flexibility to use several different window types, not

all of which are editable, the current stack is not always the same as the active window. The current stack is the target of menu choices such as **View -> Go Next** and is the stack specified by the expression *this stack*.

For example, executing the **find** command may have unexpected results if stacks of different modes are open, because under these conditions, the search may target a stack that is not the frontmost window.

Finding the current stack

The current stack--the stack that responds to commands--is designated by the **defaultStack** property. To determine which stack is the current stack, use the following rules:

1. If any stacks are opened in an editable window, the current stack is the frontmost unlocked stack. (A stack is unlocked if its **cantModify** property is set to false.)

2. If there are no unlocked stacks open, the current stack is the frontmost locked stack in an editable window.

3. If there are no stacks open in an editable window, the current stack is the frontmost stack in a modeless dialog box.

4. If there are no editable or modeless windows open, the current stack is the frontmost palette.

Another way of expressing this set of rules is to say that the current stack is the frontmost stack with the lowest **mode** property. You can find out which stack has the lowest **mode** using the **topStack** function.

The topStack function and the defaultStack property:

The **defaultStack** property specifies which stack is the current stack. By default, the **defaultStack** is set to the **topStack**, although you can change the **defaultStack** to any open stack.

The **topStack** function goes through the open stacks first by **mode**, then by layer. For example, if any editable windows are open, the topmost editable window is the **topStack**. If there are no editable windows, the

topStack is the topmost modeless dialog box, and so on.

Changing the current stack

To operate on a stack other than the current stack, set the **defaultStack** property to the stack you want to target before executing the commands. Usually, the **defaultStack** is the **topStack**, but you can change it if you want to override the usual rules about which window is active.

A note about Unix systems

If your system is set up to use pointer focus rather than click-to-type or explicit focus, you may experience unexpected results when using LiveCode, since the current stack will change as you move the mouse pointer. It is recommended that you configure your system to use explicit focus when using LiveCode or any other applications created in LiveCode.

Creating a backdrop

For some applications, you may want to create a solid or patterned backdrop behind

your application's windows. This backdrop prevents other applications' windows from being seen – although it does not close those windows – so it's appropriate for applications like a game or kiosk, where the user doesn't need to see other applications and where you want to keep distractions to a minimum.

> **Note:** In LiveCode Media edition, you cannot turn off the backdrop property.

To create a backdrop, you set the **backdrop** property to either a valid color reference, or the **ID** of an image you want to use as a tiled pattern:

```
set the backdrop to
"#99FF66" *-- a
color*
set the backdrop to
1943 *-- an image
ID*
```

In the LiveCode development environment, you can create a backdrop by choosing **View -> Backdrop**. Use the

Preferences dialog box to specify a backdrop color to use.

Open, Closed, and Hidden Windows

Each open stack is displayed in a stack window. A stack can be open without being visible, and can be loaded into memory without being open.

Hidden stacks

A stack window can be either shown or hidden, depending on the stack's **visible** property. This means a window can be open without being visible on the screen.

> **Tip:** To list all open stacks, whether they're visible or hidden, use the **openStacks** function.

Loaded stacks

A stack can also be loaded into memory without actually being open. A stack whose window is closed (not just hidden) is not listed by the **openStacks** function.

However, it takes up memory, and its objects are accessible to other stacks. For example, if a closed stack that's loaded into memory contains a certain image, you can use the image as a button icon in another stack.

A stack can be loaded into memory without being open under any of the following conditions:

A handler in another stack referred to a property of the closed stack. This automatically loads the referenced stack into memory.

The stack is in the same stack file as another stack that is open.

The stack was opened and then closed, and its **destroyStack** property is set to false. If the **destroyStack** property is false, the stack is closed, but not unloaded, when its window is closed.

> **Tip:** To list all stacks in memory, whether they're open or closed, use the **revLoadedStacks** function.

The states of a stack

A stack, then, can be in any of four states:

Open and visible: The stack is loaded into memory, its window is open, and the window is visible.

Open and hidden: The stack is loaded into memory, its window is open, but the window is hidden. The stack is listed in the Window menu and in the Project Browser.

Closed but loaded into memory: The stack is loaded into memory, but its window is not open and it is not listed by the **openStacks** function or in the Window menu. However, its objects are still available to other stacks, and it is listed in the Project Browser. A stack that is closed but loaded into memory has a **mode** property of zero.

To remove such a stack from memory, choose **Tools -> Project Browser**, find the stack's name, and Right-click it in the Project Browser window and choose "Close and Remove from Memory" from the contextual menu.

Closed: The stack is not loaded into memory and has no effect on other stacks.

Window Types and the Mode Property

In a script, you can find out a stack window's type by checking the stack's **mode** property. This read-only property reports a number that depends on the window type. For example, the **mode** of an editable window is 1, and the **mode** of a **palette** is 4.

You can use the **mode** property in a script to check what sort of window a stack is being displayed in:

```
if the mode of this
stack is 5 then *-
- modal dialog
box*
    close this
stack
else -- some
other type of
window
    beep
end if
```

For complete information about the possible values of the **mode** property, see its entry in the LiveCode Dictionary.

Window Appearance

Details of a window's appearance, such as the height of its title bar and the background color or pattern in the window itself, are mainly determined by the stack's mode. There are a few additional elements of window appearance that you can control with specific properties.

The metal property

On OS X systems, you use a stack's **metal** property to give the stack window a textured metal appearance.

This metal appearance applies to the stack's title bar and its background.

> **Tip:** The metal appearance, in general, should be used only for the main window of an application, and only for windows that represent a physical media device such as a CD player. See Apple's Aqua user-interface guidelines for more information.

Window background color

The background color or pattern of a window's content area--the part that isn't part of the title bar--is determined by the window type and operating system, by default. For example, on Mac OS X systems, a striped background appears in palettes and modeless dialog boxes.

If you want a window to have a specific color or pattern, you can set the stack's **backgroundColor** or **backgroundPattern** property:

```
set the
backgroundColor of
stack "Alpha" to
"aliceblue"
set the
backgroundPattern
of stack "Beta" to
2452 *-- img ID*
```

This color or pattern overrides the usual color or background pattern.

The Decorations Property

Most of the properties that pertain to a window's appearance can also be set in the stack's **decorations** property. The **decorations** of a stack consists of a comma-separated list of decorations:

```
set the decorations
of stack "Gamma" to
"title,minimize"
```

The statement above sets the stack's**minimizeBox** property to true, as well as showing its title bar, and sets other stack properties (**maximizeBox**, **closeBox**, **metal**) to false. Conversely, if you set a stack's **minimizeBox** property to

true, its **decorations** property is changed to include "minimize" as one of its items. In this way, the **decorations** property of a stack interacts with its **closeBox**, **minimizeBox**, **zoomBox**, **metal**, **shadow**, and **systemWindow** properties.

The decorations property and menu bars in a window

On Linux and Windows systems, the menu bar appears at the top of the window. On these systems, whether a window displays its menu bar is determined by whether the stack's **decorations** property includes "menu":

```
set the decorations
of this stack to
"title,menu"
```

On Mac OS X systems, the menu bar appears at the top of the screen, outside any window. On these systems, the "menu" decoration has no effect.

Title bar

The user drags the title bar of a window to move the window around the screen. In general, if the title bar is not displayed, the user cannot move the window. You use the **decorations** property (discussed below) to hide and show a window's title bar.

When the user drags the window, LiveCode sends a **moveStack** message to the current card.

The **decorations** property affects only whether the window can be moved by dragging it. Even if a stack's **decorations** property does not include the title bar decoration, you can still set a stack's **location**, **rectangle**, and related properties to move or resize the window.

Window title

The title that appears in the title bar of a window is determined by the stack's **label** property. If you change a stack's **label** in a script, the window's title is immediately updated.

If the **label** is empty, the title bar displays the stack's **name** property. (If the stack

is in an editable window whose **cantModify** is false, an asterisk appears after the window title to indicate this, and if the stack has more than one card, the card number also appears in the window title. These indicators do not appear if the stack has a **label**.)

Because the window title is determined by the stack's **label** property instead of its **name** property, you have a great deal of flexibility in changing window title. Your scripts refer to the stack by its **name**--which doesn't need to change--not its **label**, so you can change the window title without changing any scripts that refer to the stack.

The close box

The close box allows the user to close the window by clicking it. To hide or show the close box, you set the stack's **closeBox** property:

```
set the closeBox of
stack "Bravo" to
false
```

When the user clicks the close box, LiveCode sends a **closeStackRequest** message, followed by a **closeStack** message, to the current card.

The **closeBox** property affects only whether the window can be closed by clicking. Even if a stack's **closeBox** property is false, you can still use the **close** command in a handler or the message box to close the window.

The minimize box or collapse box

The terminology and behavior of this part of the title bar varies depending on platform. The minimize box shrinks the window to a desktop icon.

To hide or show the minimize box or collapse box, you set the stack's **minimizeBox** property:

```
set the minimizeBox
of this stack to
true
```

> **Tip:** On OS X and Linux systems, you can set a stack's **icon** property to specify the icon that appears when the stack is minimized.

When the user clicks the minimize box or collapse box, LiveCode sends an **iconifyStack** message to the current card.

The maximize box or zoom box

The terminology and behavior of this part of the title bar varies depending on platform. On Mac OS X systems, the zoom box switches the window between its current size and maximum size. The maximize box (Linux and Windows systems) expands the window to its maximum size.

To hide or show the zoom box or maximize box, you set the stack's **zoomBox** property:

```
set the zoomBox of
stack "Hello" to
false
```

When the user clicks the zoom box or maximize box, LiveCode sends a **resizeStack** message to the current card.

Making a stack resizable

A stack's **resizable** property determines whether the user can change its size by dragging a corner or edge (depending on operating system) of the stack window.

> **Tip:** To move and resize controls automatically to fit when a stack is resized, use the "Geometry" pane in the control's property inspector.

Some stack modes cannot be resized, regardless of the setting of the stack's **resizable** property. Modal dialog boxes, sheets, and drawers cannot be resized by the user, and do not display a resize box.

The **resizable** property affects only whether the window can be resized by

dragging a corner or edge. Even if a stack's **resizable** property is set to false, you can still set a stack's **location**, **rectangle**, and related properties to move or resize the window.

When the user resizes a stack, LiveCode sends a **resizeStack** message to the current card.

Irregularly-Shaped and Translucent Windows

You can set a stack's **windowShape** property to the transparent, or *alpha channel* of an image that has been imported together with its alpha channel. This allows you to create a window with "holes" or a window with variable translucency. You can apply a shape to any type of stack, regardless of the mode it is opened, allowing such a window to exhibit modal behavior as a dialog, float as a palette, etc.

You may use either a GIF or PNG image for irregularly shaped windows. If you want translucency you must use PNG images.

Programming Menus & Menu Bars

Menus in LiveCode are not a separate object type. Instead, you create a menu from either a button or a stack, then use special commands to display the menu or to include it in a menu bar.

This topic discusses menu bars, menus that are not in the menu bar (such as contextual menus, popup menus, and option menus), how to make menus with special features such as checkmarks and submenus, and how to use a stack window as a menu for total control over menu appearance.

To easily create menu bars that work cross-platform, choose **Tools -> Menu Builder**. See the section on the *Menu Builder* in the chapter on *Building a User interface* for more details. The details about menu bars in this topic are needed only if you want edit menu bars by script, for example if you

want to include specific features not supported by the Menu Builder.

Menu Types

LiveCode supports several menu types: *pulldown* menus, *option* menus (usually called popup menus on Mac OS X), *popup* menus (usually called contextual menus on Mac OS X), and *combo boxes*.

Each of these menu types is implemented by creating a button. If the button's **style** property is set to "menu", clicking it causes a menu to appear. The button's **menuMode** property determines what kind of menu is displayed.

Even menu bars are created by making a pulldown-menu button for each menu, then grouping the buttons to create a single menu bar. The menu bar can be moved to the top of the stack window (on Unix and Windows systems). To display the menu bar in the standard location at the top of the screen on Mac OS X systems, you set the stack's **menubar** property to the

group's name. The name of each button is displayed in the menu bar as a menu, and pulling down a menu displays the contents of the button as a list of menu items.

Button Menus

You can create a button menu by dragging out one of the menu controls from the tools palette. However, if you want to create one by script, the easiest is to create a button and set the **style** of the button to "menu". Next, you can set the **menuMode** of the button to the appropriate menu type. You can either set the **menuMode** in a handler, or use the Type menu in the button's property inspector to set the menu type.

To create the individual menu items that will appear in the menu, set the button's **text** property to the menu's contents, one menu item per line. You can either set this property in a handler, or fill in the box labeled "Menu items" on the Basic Properties pane of the property inspector.

When you click the button, the specified menu type appears, with the text you entered displayed as the individual menu items in the menu.

> **Tip:** To dynamically change the menu's contents at the time it's displayed, put a **mouseDown** handler in the button's script that puts the desired menu items into the button. When the menu appears, it displays the new menu items.

For menus that retain a state (such as option menus and combo boxes), the button's **label** property holds the text of the currently chosen menu item.

Handling the menuPick message

When the user chooses an item from the menu, LiveCode sends the **menuPick** message to the button. The message parameter is the name of the menu item chosen. If you

want to perform an action when the user chooses a menu item, place a **menuPick** handler like this one into the button's script:

```
on menuPick
theMenuItem
    switch
theMenuItem
    case "Name of
First Item"
        -- do
stuff here for
first item
    break
    case "Name of
Second Item"
        -- do
stuff here for
second item
    break
    case "Name of
Third Item"
        -- do
stuff here for
third item
    break
    end switch
end menuPick
```

Changing the currently-chosen menu item

For menus that retain a state (such as option menus and combo boxes), you can change the currently-chosen menu item by changing the button's **label** property to the text of the newly chosen item

If you change the currently-chosen menu item in option menus, also set the button's

menuHistory property to the line number of the newly chosen item. This ensures that the new choice will be the one under the mouse pointer the next time the user clicks the menu.

Creating Cascading Menus

To create a cascading menu (also called a submenu, pull-right menu, or hierarchical menu), add a tab character to the start of menu items that you want to place in the submenu.

For example, the following text, when placed in a menu button, creates two menu items, then a submenu containing two more items, and finally a last menu item:

First Item

Second Item

Third Item Is A Submenu

First Item In Submenu

Second Item In Submenu

Last Menu Item Not In Submenu

The depth of a submenu item is determined by the number of tab characters before the

menu item's name. The submenu item becomes part of the closest line above the submenu item that has one fewer leading tab character.

This means that the first line of a menu cannot start with a tab character, and any line in the button's text can have at most one more tab character than the preceding line.

> **Important:** You cannot create a cascading combo box at all, and cascading option menus do not work properly on all platforms. In general, you should create cascading menus only as a part of a pulldown menu.

Cascading menus and the menuPick message

When the user chooses a menu item in a cascading menu, the parameter of the **menuPick** message contains the menu item name and the name of the submenu it's part of, separated by a vertical bar (`|`). For

example, if the user chooses the "Second Item In Submenu" from the menu described above, the parameter sent with the **menuPick** message is:

```
Third Item Is A Submenu|Second Item In Submenu
```

Ticks, Dashes & Checks in Menus

There are several special characters that you can put at the start of a line in the button's contents to change the behavior of the menu item:

- A dash on a line by itself creates a divider line !c checks the menu item !n unchecks the menu item !r places a diamond at the start of the menu item !u removes the diamond

If you include any of the above special characters in a submenu item, the special character must be placed at the start of the line – before the tab characters that make it a submenu item.

> **Note:** You cannot create divider lines in combo boxes or in option menus on Windows systems.

There are three other special characters that can appear anywhere in a line:

Putting the `&` character anywhere in a line underlines the next character and makes it the keyboard mnemonic for that menu item on Windows systems. The `&` character does not appear in the menu, and is not sent with the parameter to the **menuPick** message when you choose the item from a menu.

Putting the `/` character anywhere in a line makes the next character the keyboard equivalent for the menu item. Neither the `/` nor the character following it appear in the menu, nor do they appear in the parameter to the **menuPick** message.

To put an `&` or / character in the text of a menu, double the characters: & `&` or //.

Putting the `(` character anywhere in a line disables the menu item. To put a `(` character in a menu item without disabling it, precede it with a backslash: (.

> **Note:** You cannot disable lines in combo boxes or in option menus on Windows systems.

All of the above special characters are filtered out of the parameter sent with the **menuPick** message when the user chooses a menu item. The parameter is the same as the characters that are actually displayed in the menu.

> **Note:** The font and color of a button menu is determined by the button's font and color properties. However, on Mac OS systems, the font and color of the option menus and popup menus is controlled by the operating system's settings if the **lookAndFeel** is set to "Appearance Manager", rather than by the button's font and color properties.

Enabling and disabling menu items

To enable or disable a menu item in a handler, you can add or remove the "(" special character, but it is generally easier to use the **enable menu** and **disable menu** commands:

```
enable menuItem 3
of button "My Menu"
disable menuItem 4
of me
```

These commands simply add or remove the `(` special character at the start of the designated line of the button's contents.

Menu Bars on Linux and Windows Systems

A menu bar is made up of a group of menu buttons, with the **menuMode** property of each button set to "pulldown".

> **Tip:** The Menu Builder can automatically create a menu bar for you. To use the Menu Builder, choose **Tools -> Menu Builder**.

To create a menu bar by hand without using the Menu Builder:

1. Create a button for each menu, set the **style** of each button to "menu", and set the **menuMode** of the button to "pulldown". You can either set these properties in a handler, or simply choose **Object -> New Control Pulldown Menu** to create each button.

2. Put the menu items into each button's contents. In each button's script, create a **menuPick** handler to perform whatever actions you want to do when a menu item is chosen.

3. Select the buttons and form them into a group, then move the group to the appropriate position at the top of the window. For Windows systems, set the **textFont** of the group to the standard font for Windows menus, "MS Sans Serif".

> **Important:** The buttons in your menu bar should not overlap. Overlapping buttons may cause unexpected behavior when the user tries to use a menu.

Menu Bars on Mac OS X Systems

To create a Mac OS X menu bar, you follow the same steps as for a Linux and Windows menu bar above. This places a group of buttons, each of whose

menuMode property is set to "pulldown", at the top of your stack window.

Next, you set the **menubar** property of your stack to the name of the group. This does two things: it displays the menus in the menu bar at the top of the screen, and it shortens the stack window and scrolls it up so that the group of menu buttons is not visible in the window. Since the menus are in the menu bar, you don't need to see them in the stack window as well.

> **Important:** If your stack has more than one card, make sure that the group is placed on all the cards. (To place a group on a card, choose **Object menu Place Group**, or use the **place** command.) This ensures that the menu bar will be accessible on all cards of the stack, and prevents the stack from changing size as you move from card to card (to accommodate shortening the stack window for the menu bar group).

The default menu bar

If other stacks in your application don't have their own menu bars, set the **defaultMenubar** global property to the name of your menu group, as well as setting the stack's **menubar** property. The **defaultMenubar** is used for any stack that doesn't have a menu bar of its own.

> **Tip:** For a custom menu bar to work correctly inside the LiveCode development environment, you must set the **defaultMenubar** to the name of your menu group. This overrides the LiveCode IDE menu bar. You can get the menu bar back by choosing the pointer tool.

Button menu references

If the button is a button menu that's being displayed in the menu bar, you can use the word "menu" to refer to it:

```
get menuItem 2 of
menu "Edit"
-- same as 'get
line 2 of button
"Edit"'
```

Because menus are also buttons, you can use a button reference to get the same information. But you may need to specify the group and stack the button is in, to avoid ambiguity. (For

example, if there is a standard button named "Edit" on the current card, the expression `button "Edit"` refers to that button, not to the one in the menu bar.) An unambiguous button reference to a menu might look like this:

```
get line 2 of
button "Edit" of
group "Menu" of
stack "Main"
```

The above statement produces the same information as the form using "`menu`", but you need to know the group name and possibly which stack it's in, so the *menuName* form is a little more convenient.

The layer of menu buttons

For a menu bar to work properly on Mac OS X systems, the menus must be in layer order within the group. That is, the button for the File menu must be numbered 1, the button for the Edit menu must be 2, and so on. The Menu Builder takes care of this automatically; you only need

to worry about layering if you're creating the menu bar by hand.

Changing menus dynamically

If you want to dynamically change a menu's contents with a **mouseDown** handler at the time the menu is displayed, you must place the **mouseDown** handler in the group's script. When a menu button is being displayed in the Mac OS menu bar, it does not receive **mouseDown** messages, but its group does.

The editMenus property

When you set the **menubar** property of a stack to the name of a group, the stack is resized and scrolled up so the part of the window that holds the menus is not visible. To reverse this action so you can see, select and edit the buttons that make up your menu bar, set the **editMenus** property to true. This resizes the stack window so the button menus are again visible, and you can use the tools in the LiveCode development environment to make changes to them.

To scroll the stack window again so that the menus are hidden, set the **editMenus** property back to false.

Special menu items

A few menu items on Mac OS X are handled directly by the operating system. To accommodate these special menu items while allowing you to create a fully cross-platform menu bar, LiveCode treats the last two menu items of the Help menu, the File menu, and the Edit menu differently.

By following these guidelines, you can make sure your menus will appear properly on all operating systems without having to write special code or create platform-specific menu bars.

The Help menu and the "About This Application" menu item

When LiveCode sets up the Mac OS X menu bar, it automatically makes the last button the Help menu (regardless of the button's name). The standard Help menu items, such as the "Search" bar are included for you automatically; you don't

need to include them in your Help menu button, and you can't eliminate them from the Help menu.

LiveCode moves the last menu item in the Help menu to the "About This Application" position. On Mac OS X systems, it's the first menu item in the Application menu. Therefore, the last menu item in your Help menu button should be an appropriate "About" item. The menu item above it must be a divider line (a dash), and above that must be at least one menu item to be placed in the Help menu.

The File menu and the "Quit" menu item

On Mac OS X systems, the "Quit" menu item is normally placed in the Application menu (which is maintained by the operating system) rather than in the File menu, as is standard on other platforms. To accommodate this user-interface standard, LiveCode removes the last two menu items of the File menu when a standalone application is running on an OS X system. Therefore, the last menu item in your File menu button

should be "Quit". The menu item above it should be a divider line (a dash).

The Edit menu and the "Preferences" menu item

On OS X systems, the "Preferences" menu item is also normally placed in the Application menu. To accommodate this user-interface standard, LiveCode removes the last two menu items of the Edit menu when a standalone application is running on an OS X system. Therefore, the last menu item in your Edit menu button should be "Preferences". The menu item above it should be a divider line (a dash).

> **Note:** The Preferences menu item is treated in this special way only if its name starts with the string "Preferences".

> **Tip:** If your application's user interface is presented in a language other than English, set the **name** of the Edit menu button to "Edit", and set its **label** to the correct translation. This ensures that the engine can find the Edit menu, while making sure that the menu is shown in the correct language.

Choosing the special menu items

When the user chooses any of these special menu items, a **menuPick** message is sent to the button that the menu item is contained in. This ensures that your button scripts will work on all platforms, even if LiveCode displays a menu item in a different menu to comply with user-interface guidelines.

Stack Menus

Button menus can be used for most kinds of standard menus. However, if you want

to create a menu with a feature that is not supported by button menus--for example, if you want a popup menu that provides pictures, rather than text, as the choices--you can create a menu from a stack.

Creating a stack menu

To create a stack menu, you create a stack with a control for each menu item. Since the stack menu is a stack and each menu item is an object, the menu items receive mouse messages such as **mouseEnter**,**mouseLeave**, and **mouseUp**.

When the user chooses an item from the stack menu, a **mouseUp** message is sent to that control. To respond to a menu item choice, instead of handling the **menuPick** message, you can place a **mouseUp** handler in the script of the object.

To create a stack menu that looks like a standard menu, create a button in the stack for each menu item. The button's **autoArm** and **armBorder** properties should be set to true. Or you can choose "Menu Item" item in

the "New Control" submenu of the Object menu to create a button with its properties set to the appropriate values.

Be sure to set the **rectangle** of the stack to the appropriate size for the menu. Remember, when you open the menu, the stack will be displayed exactly as it looks in an editable window.

Finally, either set the **menuName** property of a button to a reference to the stack, or place a **mouseDown** handler containing a **pulldown**, **popup**, or **option** command in the script of an object. When you click the button or object, the stack menu appears.

Displaying a stack menu

Stack menus can be associated with a button, just like button menus. But when you click the button, instead of displaying a menu with the button's contents, LiveCode displays a stack with the behavior of a menu.

You can also display a stack menu without associating it with a button, by using the **pulldown**, **popup**, or **option**

command. Normally, you use these commands in a **mouseDown** handler, so that the menu appears under the mouse pointer:

```
on mouseDown -- in card script
    popup stack "My Menu Panel"
end mouseDown
```

Displaying Context Sensitive Menus

There are also several commands to display a context menu. Usually, you use these commands in a **mouseDown** handler – normally either in your card or stack script:

popup command: opens a stack as a popup menu

pulldown command: opens a stack as a pulldown menu

option command: opens a stack as an option menu

> **Note:** If you set a button's **menuName** property to the name of a stack, the stack menu is displayed automatically when the user clicks the button. You need the **popup**, **pulldown**, and **option** commands only if you want to display a stack menu in some way other than when a button is clicked.

Searching and Navigating Cards using the Find Command

The find command in LiveCode allows you to search the fields of the current stack, then navigate to and highlight the results of the search automatically. While it is possible to build such a command using the comparison features detailed in the *Processing Text and Data* guide, for most purposes the find command provides a complete, pre-built solution.

find [*form*] textToFind [*in field*]

The *form* can be one of the following:

normal

characters or character (or chars or char)

words or *word*

string

whole

If no *form* is specified, the find normal form is used.

The *textToFind* is any expression that evaluates to a string.

The *field* is any expression that evaluates to a field reference. If the *field* is not specified, the **find** command searches all the fields in the current stack (except fields whose dontSearch property is set to true).

```
find "heart"
find string "beat
must go on" in
field "Quotes"
```

When the find command finds a match, it highlights the match on the screen – if necessary navigating to the

card that contains the match and scrolling the field so the text is in view.

The find command can also be used to return the location of the text that was found.

To reset the find command so that it starts searching at the beginning again:

```
find empty
```

For more details on the find command and associated options, see the *find command* in the *LiveCode Dictionary*.

Using Drag and Drop

LiveCode allows you complete control over drag and drop – both within LiveCode windows and between LiveCode and other applications.

Initiating a Drag Drop

To begin a drag and drop operation, the user clicks and holds the mouse pointer. This sends a `mouseDown` message to the object.

If you drag from within a field, a **dragStart** message is sent. To allow drags from a locked field or from another object type, in the object's **mouseDown** handler, set the **dragData** property to the data you want to drag. When there is a value in the **dragData**, a drag and drop is initiated when the mouse is clicked and then moved.

```
set the
dragData["text"] to
"text being
dragged"
```

You can set the **dragData** to contain any of the following types of data:

text	The plain text being dragged.
HTML	The styled text being dragged, in the same format as the **htmlText**
RTF	The styled text being dragged, in the same format as the **RTFText**
Unicode	The text being dragged, in the same format as the **unicodeText**

image	The data of an image (in PNG format)
files	The name and location of the file or files being dragged, one per line

> **Note:** LiveCode automatically handles the mechanics of dragging and dropping text between and within unlocked fields. To support this type of drag and drop operation, you don't need to do any scripting.

For more details, see the entries for *dragStart* and *dragData* in the *LiveCode Dictionary*.

Tracking During a Drag Drop Operation

You can use the **dragEnter** message to show an outline around an object or change the cursor when the mouse moves into it during a drag operation.

```
on dragEnter --
show a green
outline around
the drop target
    set the
borderColor of the
target to "green"
end dragEnter
```

You can use the **dragMove** message to update the screen whenever the cursor moves during a drag and drop operation.

```
on dragMove -- in
a field script
    -- set the
cursor so it
shows you can
only drop onto a
link
    if the
textStyle of the
mouseChunk contains
"link"
    then set the
cursor to the ID of
image "Drop Here"
    else set the
cursor to the ID of
image "Dont Drop"
end dragMove
```

You can use the **dragLeave** message to remove any outline around an object or change the cursor when the mouse moves out of an object during a drag operation.

```
on dragLeave
    -- remove any
outline around
the drop no-
longer-target
    set the
borderColor of the
target to empty
end dragLeave
```

For more details, see the entries for *dragEnter*, *dragMove* and *dragLeave* in the *LiveCode Dictionary*.

Responding to a Drag and Drop

To perform an action when the user drops data onto a locked field or another object type, you handle the **dragDrop** message.

The **dragDrop** message is sent when the user drops data on an object.

```
on dragDrop --
check whether a
file is being
dropped
    if the
dragData["files"]
is empty then beep
2
    pass dragDrop
end dragDrop
```

You must set the **acceptDrop** property to true before a drop will be allowed. Usually, you set this property to true in a **dragEnter** handler.

You can use the **dragDestination** function to retrieve the long id of the object that the dragged data was dropped on. You can use the **dragSource** function to retrieve the long id of the object that was the source of the drag.

When a drag drop has been completed, a **dragEnd** message is sent to the object the drag and drop started from.

```
on dragEnd --
remove data being
dragged
d   elete the
dragSource
end dragEnd
```

You can use the **dropChunk** function to retrieve the location of the text that was dropped in a field. For example, you could select the text that was dropped by doing the following:

```
select the
dropChunk
```

For more details, see the entries for *dragDrop*, *dragEnter*, *dragDestination*, *dragEnd*, *dragSource* , *dropChunk* and *acceptDrop* in the *LiveCode Dictionary*.

Prevent Dragging and Dropping to a Field

You prevent dropping data into a field during a drag and drop by setting the **acceptDrop** property to false when the mouse pointer enters the field.

If the **acceptDrop** is set to false, when you drop data, no **dragDrop** message is sent to the field. Since the drop is automatically processed only when LiveCode receives a **dragDrop** message, this prevents the usual automatic drop behavior.

Usually, you should set the **acceptDrop** in a **dragEnter** handler, as in the following example:

```
on dragEnter -- in
a field script
    set the
acceptDrop to false
end dragEnter
```

If you want to prevent dragging text within a field, intercept the dragStart message:

```
on dragStart
    -- do nothing
end dragStart
```

For more details, see the entries for *acceptDrop*, *dragDrop* and *dragEnter* in the *LiveCode Dictionary*.