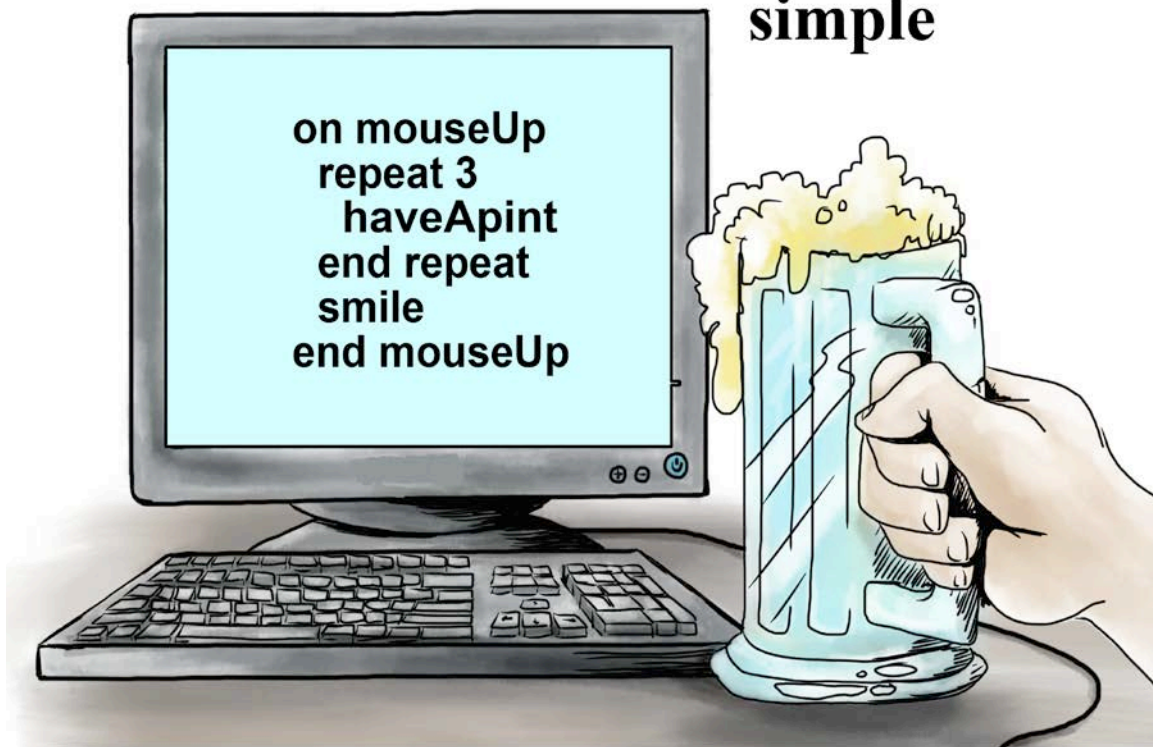


LiveCode Lite

Computer Programming
made
ridiculously
simple



Stephen Goldberg

LiveCode Lite: Computer Programming Made Ridiculously Simple

**Stephen Goldberg, M.D.
Professor Emeritus
University of Miami School of Medicine**

MedMaster Inc., Miami FL



Copyright © 2015 by MedMaster Inc.

All rights reserved. This book is protected by copyright. No part of it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the copyright owner.

ISBN 978-1-935660-21-7

Made in the United States of America

Published by
MedMaster, Inc.
P.O.Box 640028
Miami, FL 33164

CONTENTS

SECTION 1. LIVECODE BASICS

- Preface
- Chapter 1. Introduction
- Chapter 2. The Stack Metaphor
 - Kinds of Stacks
 - The Message Box
- Chapter 3. The Tools Palette
 - Buttons
 - Fields
 - Menu Objects
 - Scrollbars
 - Image and Quicktime Controls
 - Draw and Paint Tools
- Chapter 4. Groups
- Chapter 5. The Application Browser
- Chapter 6. The Message Flow Hierarchy

SECTION 2. SCRIPTING

- Chapter 7. Mouse-related Script Words
- Chapter 8. Navigation Commands
- Chapter 9. General Action Commands
- Chapter 10. Keyboard Script Words
- Chapter 11. Variables and Custom Properties
- Chapter 12. Me vs. The Target
- Chapter 13. Functions
- Chapter 14. Math
- Chapter 15. Constants
- Chapter 16. If-Then-Else and Repeat Structures
- Chapter 17. Cursors
- Chapter 18. Printing
- Chapter 19. Internet Communication
- Chapter 20. Special Effects Scripting
- Chapter 21. Script Debugging

SECTION 3. PROPERTY INSPECTORS

- Chapter 22. Stack Property Inspector
 - Stack Scripting
- Chapter 23. Card Property Inspectors
 - Card Scripting
- Chapter 24. Button Property Inspector
- Chapter 25. Menu Property Inspectors
 - Menu Scripting
- Chapter 26. Field Property Inspector
 - Field Scripting

Chapter 27. Image Property Inspector
Chapter 28. Player & AudioClips Property Inspectors
Chapter 29. Absolute vs. Referenced File Paths
Chapter 30. Graphics Property Inspectors
Chapter 31. Scrollbar/Slider/Little Arrows/Progress Bar Property
Inspectors
Chapter 32. Multiple Objects Property Inspector
Chapter 33. Groups Property Inspector

SECTION 4. THE MENU BARS

Chapter 34. The LiveCode Menu Bars

REFERENCES

SECTION 1. LIVECODE BASICS

CHAPTER 1. INTRODUCTION

What Is LiveCode?

LiveCode (formerly called Runtime Revolution) is an intuitive and powerful programming environment with a short learning curve, in which programming is done in simple English, with rapid results.

Whether in business, education, or game development, many people do not have the resources to hire an IT person or the time to learn programming languages with steep learning curves. LiveCode is an extremely versatile program, where in very little time one can create advanced applications with text, interactive buttons, images and vector graphics, movies, sounds, and Internet and database connectivity. With a click of a button, your creations, whether developed on the Macintosh or Windows versions of LiveCode, can be built for Macintosh, Windows, Linux and mobile devices. Educators who are looking for a simple but powerful way to create applications for their classes, or teach students to program, will find LiveCode an excellent way to create even complex teaching materials, or to teach the general principles of programming to their classes.

LiveCode, while easy to use, is a professional tool that gives developers a competitive edge in the speed at which they can complete their projects in comparison with program environments such as Java and C++.

LiveCode is an outgrowth of Apple's HyperCard program of the late 1980s, but is far more powerful. While the original HyperCard language contained about 150 programming words, LiveCode contains close to 2000 and many features far more advanced than the original HyperCard. LiveCode has been described as "HyperCard on steroids".

With so many advanced features, and new ones every year, it can be difficult for the beginner to get started with the program. While some people can learn LiveCode from scattered tutorials, others need a brief, linear, step-by-step approach, as emphasized in this book.

This is *not* a reference text or a book for the expert. It covers only the most basic aspects of LiveCode to enable the beginner to see the forest for the trees. It does not cover features that are seldom used or advanced areas, such as databases and programming specific to mobile devices. What the book describes, though, should be useful for almost any project idea. One of the best-selling games of all

times was the point-and-click game *Myst*; it was created with HyperCard, which was far less powerful than LiveCode.

This book emphasizes programming features that I have found most useful in over 25 years of programming educational materials at the University of Miami School of Medicine and the Medmaster Publishing Company.

LiveCode is a much deeper program than can be covered in this brief book. Yet you can accomplish much with these basics.

Throughout the book, scripting words will be placed in *italics*, while key words will be in **boldface**. The pictures throughout this book are those taken with the Macintosh interface. However, the features in Windows are very similar.

I thank Jacqueline Landman Gay, who reviewed an earlier version of the manuscript, for many helpful suggestions.

Let's get started!

CHAPTER 2. THE STACK METAPHOR

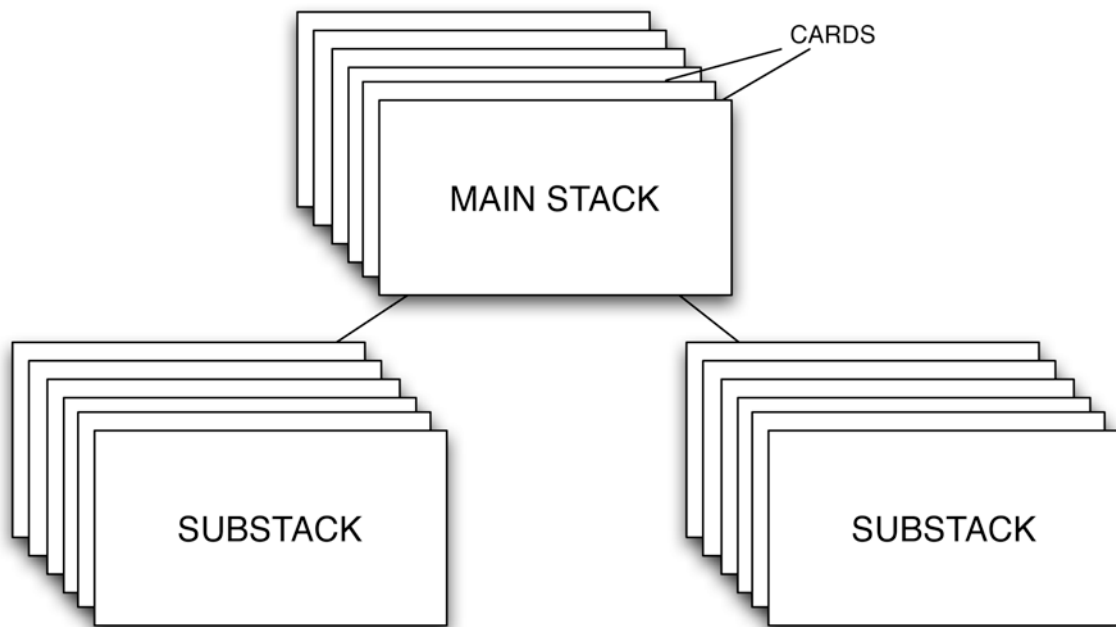


Fig. 2-1.

LiveCode uses the original HyperCard metaphor of a **stack** (deck) of **cards** (Fig. 2-1). A **stack** consists of one or more cards, on which you can place various objects, such as buttons, fields, images, and movies. Although only one card in a stack can be seen at a time (like a neatly piled deck of card), the user can

navigate from one card to another, or communicate from a card to outside sources, such as other stacks, or the Internet.

Open LiveCode by clicking on the LiveCode application icon. You don't immediately see any stacks, just a Menu Bar (**Fig. 2-2**), an Icon Tool Bar, and a Tools Palette with icons of the kinds of things you can put on a card, such as buttons, fields, images, and movies (**Fig. 2-2**). Actually, you may first see a tutorial **Get Started Center** window, which automatically opens, unless you opt to turn this feature off by unchecking the "Show this screen on Startup" box at the lower left corner of the Start Center window. To reshown this Help window, select **HELP/START CENTER**.

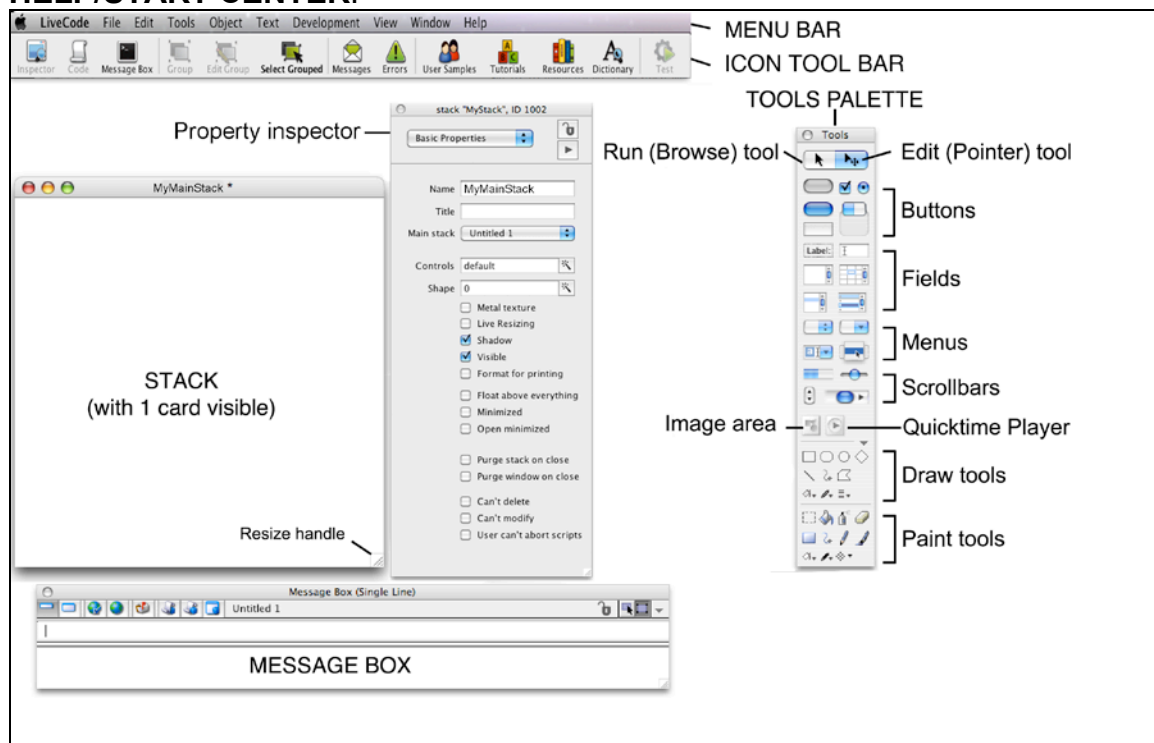


Fig. 2-2.

1. From the **Menu Bar** select **File/New Mainstack**. This creates a stack called a "main stack" that contains a single card (**Fig. 2-2**). Under the Menu Bar, there is also a quick-select **Icons Tool Bar**, with text names for the icons. If you don't see the Icons Bar or its text, select **VIEW/Toolbar Text** and **VIEW/Toolbar Icons** from the Menu Bar.

2. Name this stack after selecting **Object/Stack Inspector** and typing **MyMainstack** in the Stack Inspector's **Name** field. Alternatively, you can open the stack inspector by clicking on the **Inspector** icon in the Icon Toolbar. And still another way is to right click on the stack and choose **Stack Property Inspector**.

3. Close the **Inspector** window. Note that the name of the stack now appears at the top of the stack in the stack's title bar.

4. Save this stack (**FILE/SAVE**) to your desktop. LiveCode will by default use the mainstack's name and save the stack to your desktop as a **stack file**, called **MyMainstack.livecode**, for future use in this book.

5. Position the stack where you wish on the desktop by holding the mouse down over the stack's title bar and dragging. Resize the stack as you want by holding the mouse down and dragging the resize handle at the lower right hand corner of the stack (**Fig. 2-2**).

A **mainstack** can have associated **substacks** (**Fig. 2-1**). It is like dividing a book into chapters. You can have a long book without chapters, but it is often better to organize books with individual chapters. Similarly, while you could create just one stack with many cards (pages) in it, it is often better organized to create a mainstack with connections to one or more substacks.

Let's create a substack:

1. Select **File/New Substack of MyMainstack**.
2. Right click on this substack; select **Stack Property Inspector**.
3. Name this substack **MySubstack** and close the stack's Inspector window.
4. Save your work (**File/Save**) and, on the desktop, change your desktop file name from **MyMainstack.livecode** to **MyTutorial.livecode** (or **MyTutorial.rev** if you are using an older version of LiveCode, which was called Runtime Revolution. Note that although the desktop file now is titled **MyTutorial.livecode**, the Mainstack is still titled **MyMainstack** and the substack is still called **MySubstack**.
5. Quit LiveCode (**LIVECODE/QUIT**).
6. Now open **MyTutorial.livecode** (**MyTutorial.rev**) by double-clicking on its icon. Uh, Oh! You only see one stack, **MyMainstack**. Where is **MySubstack**? Did it disappear!? Actually, it is by default closed; only a mainstack opens when a LiveCode file is opened. Confirm that stack **MySubstack** is really there, as follows:

1. Select **Tools/Application Browser** (**Fig. 2-3**).

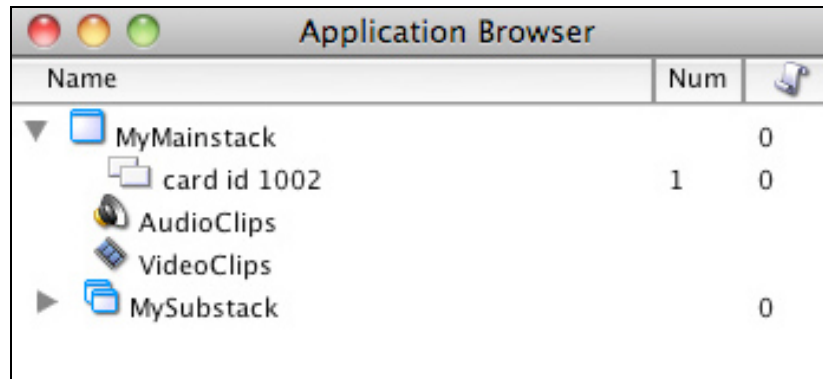


Fig. 2.3. Application Browser

2. The **Application Browser** lists all the stacks and substacks in the stack file of MyTutorial.livecode, in this case **MyMainstack** and **MySubstack**.

3. Double click on the word **MySubstack** in the Application Browser. **MySubstack** opens (so it is really there!).

Since both **MyMainstack** and **MySubstack** both have a white background, let's distinguish their appearance:

1. Open **MySubstack's** card Property Inspector by right clicking on the MySubstack card (or just double-clicking on the card) and choose **Card Property Inspector**. Name the card "Green card".

2. Choose **Colors & Patterns (Fig. 2-4)** from the Card Inspector's pulldown menu at the top of the Property Inspector. The row labeled **Background** has two boxes, the left one for patterns and the right one for colors. Click on the right-hand box and select a green color. Click "OK". **MySubstack's card** is now colored green, to distinguish it from the white **MyMainstack**.

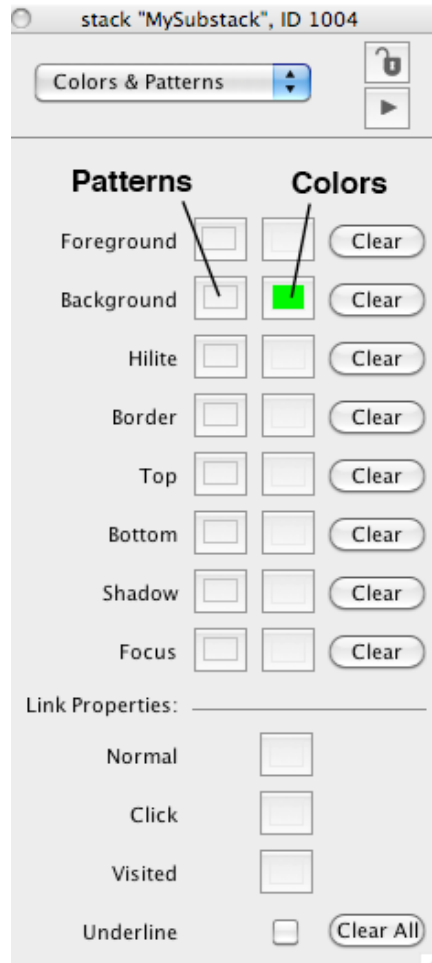


Fig. 2-4. Colors and Patterns

Note!: If you move the mouse cursor over almost any of the words or symbols in a Property Inspector, a tiny note generally appears, called a **tooltip**. For instance, just positioning the mouse over the color box that you just clicked in the Property Inspector reveals the tooltip *backgroundColor*, which can be used in a script. There will be more about scripting later, but as a quick example, if you were to write as a script:

set the backgroundColor of this card to green

this would set the color of the card to green, without having to open the Inspector to make this change. This is a powerful feature of LiveCode. You can not only set the properties of objects in the stack through their Property Inspectors, but you can, using the script word *set*, set any property of an object by script, whose words can easily be looked up through the tooltip feature.

3. Close the MySubstack Card Property Inspector. Save your work.

Stack **MySubstack** has only one card. Let's create 2 more:

1. Click on stack **MySubstack**.
2. Select **Object/New Card** from the menubar. Do this once more to create a total of 3 cards in the stack.
3. Save your work.
4. There are now 3 cards in stack **MySubstack** (one green and 2 white). But where are these new cards? Only one card in a stack is visible at a time, since cards lie directly under one another. The number of each card appears in parentheses at the top of the card in the card's titlebar. There should be number "(1)", "(2)", or "(3)" in the titlebar of MySubstack, indicating which of the three cards you are looking at. To move from one card to the next, select **View/Go Next** or **View/Go Prev** from the Menu Bar or pressing their keyboard equivalents shown on the Menu Bar. Note how the number of the card changes as **MySubstack** cycles through its 3 cards.
5. To even more clearly distinguish the three cards in stack MySubstack, name the second card "Red card" and color it red. Name the third card "Yellow card" and color it yellow. Be sure you make these color changes in each individual **card** via their card Property Inspectors, rather than to the stack Property Inspector, which would set the color to the stack as a whole. (Sometimes the stack Property Inspector just pops up on its own, which can be a nuisance, so be sure you are working with the **Card** Property Inspector). If you were to set the **stack's** color some color other than green, red, or yellow, note that setting the color of each individual card overrides any color assigned to the stack. Other card properties also override those of the stack.
6. Save your work.
7. Open the Application Browser (**Tools/Application Browser**).
8. Click on the little arrow to the left of MySubstack (**Fig. 2-5**). You should see listed under "MySubstack" the names of the cards you have created, "Green card", "Red card", " " and "Yellow card", Right click on any of these names to reveal a menu for each card. Select "go" . This opens the corresponding card.

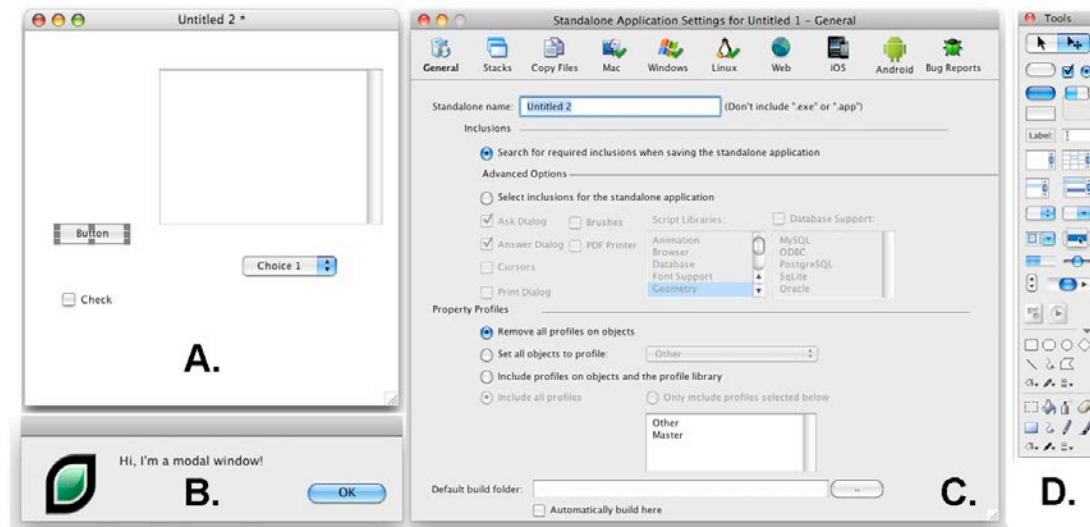
Name	Num	ID	Marked
▼ MyMainstack			0
card id 1002	1	1002	0
AudioClips			
VideoClips			
▼ MySubstack			0
Green card	1	1002	0
Red card	2	1003	0
Yellow card	3	1004	0
AudioClips			
VideoClips			

Fig. 2.5. Application Browser

9. Quit Livecode.

Kinds Of Stacks

There are 4 general types of stacks: **topLevel**, **modal**, **modeless** and **palette**, which correspond to the way a user can interact with them.



A. TopLevel stack. B. Modal stack. C. Modeless stack. D. Palette stack.

Fig. 2.6.

TopLevel stacks are the standard default type (Fig. 2-6A). This is the only fully editable stack.

Modal stacks (Fig. 2-6B) require a response from the user before any other stack can be used. They frequently are in the form of a dialog box (appearing in

response to *answer* or *ask* commands, to be discussed later under Scripting), requiring the user to input some information before the user can return to the underlying stack.

Modeless stacks (Fig. 2-6C) are like the standard `topLevel` ones, except the user is limited to typing in text in fields and clicking on buttons.

Palette stacks (Fig. 2-6D) commonly contain tools, may have your own personal icons that can be accessed for use in stack you are working on. An example of a palette stack is the Tools palette that comes with LiveCode.

Usually, you will only be creating `TopLevel` or modal stacks.

The Message Box

The **Message Box (Fig. 2-8)** is very useful in testing scripting commands. For example:

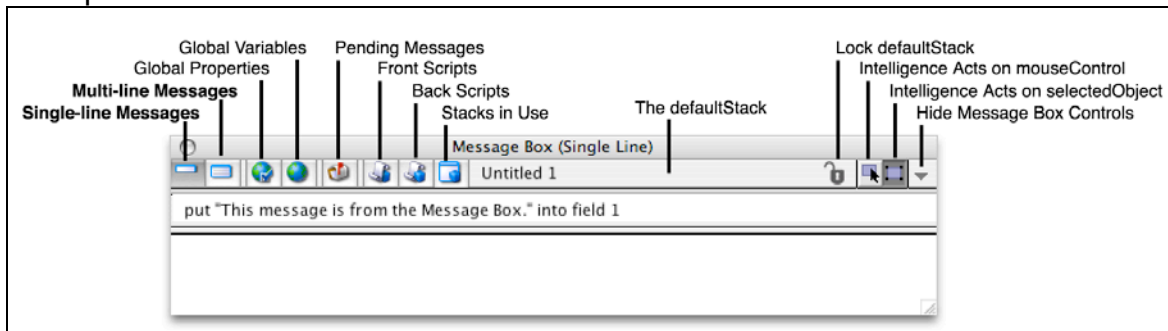


Fig. 2.7. Message Box

1. Open LiveCode and create a new mainstack (**FILE/NEW MAINSTACK**). Open the Message Box with **Tools/ Message Box**, or use the keyboard shortcut Command-M (for Macintosh) or Control-M (for Windows). (In general, keyboard combinations that use the “Command” key in Macintosh use the “Control” key in Windows.)

2. Note that each of the objects on the Tools Palette has a tooltip name that can be seen simply by placing the mouse cursor directly over the icon.

The top of the Tools Palette (**Fig. 2-2**) contains an arrow on the left, called the **Run (browse) tool** and a hatched arrow on the right, the **Edit tool (pointer) tool**. When you click on the Edit tool, you are in **Edit mode**, able to edit your stack. When you click on the Run tool, you are instantly taken to **Run mode**, where you can test how the stack runs. This easy ability to move between editing and running the program is one of the reasons that allow rapid development time in LiveCode.

Double click on the Text Entry Field in the Tools Palette to place a Text Entry Field at the center of the card (**Fig. 2-8**). Select the field by clicking on it in Edit mode with the mouse, and enlarge the field somewhat by pulling on its handles.

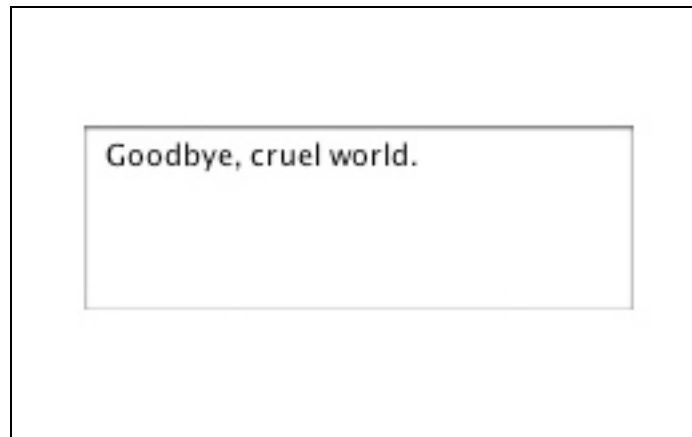


Fig. 2-8.

3. Type in the Message Box:

put "Goodbye cruel world." into field 1

Then press Return. The words "Goodbye cruel world." will appear in the field. Thus, the Message Box is a quick way to deliver and test messages.

There are a number of icons along the top of the Message Box (**Fig. 2-7**). Two are particularly helpful. The one at the very left is for **single-line** commands. It enables you to test a single line of script. After typing a message, you press Return to effect the command.

The icon second from the left enables you to test **multiple-line commands**. In multiple-line commands, your commands go into effect when you press the Enter key (not the Return key, which is needed to do carriage returns for multiple lines). Click on the **Multiple-line** icon of the Message box and type the following:

put "This message is from the multiple-line Message Box." into field 1
beep

Then press the Enter key. The words will appear in the field, followed by a beep - multiple commands.

You can also issue multiple commands in the single-line Message Box by inserting a ";" between the commands. Thus:

put "This message is from the single-line Message Box" into field 1;beep

If you can't remember what previous commands you wrote in the single-line Message Box, LiveCode automatically remembers them. Simply press the up key arrow, which will scroll you through previous commands and enable you to use them again. Be careful not to use objectionable language in your command, as someone using your stack will be able to find them using the up arrow!

Quit LiveCode. There is no need to save your work (unless you want to pin it up on the refrigerator).

CHAPTER 3.

THE TOOLS PALETTE

Open the stack file MyTutorial.livecode (MyTutorial.rev). You should see a white (Mainstack) card and the Tools Palette.

The **Tools Palette (Fig. 2-2)** contains icons of the various **controls** (also called **objects**) that can be placed on a card. (Actually, the term “objects” is a little broader than “controls”, as “objects” includes cards and stacks as well as controls. But we will not fuss over the difference.) If the tools palette is not visible, select **Tools/Tools Palette**.

By holding the mouse over any of the Tools Palette icons, the name of the tool appears. As you can see, the tools include a variety of buttons, fields, menu objects, scrollbars, an image area control, Quicktime player, and a set of drawing and paint tools (**Fig. 2-2**).

As mentioned, the top of the Tools Palette contains an arrow on the left, called the **Run (browse) tool** and a hatched arrow on the right, the **Edit tool (pointer) tool**. When you click on the Edit tool, you are in **Edit mode**, able to edit your stack. When you click on the Run tool, you are instantly taken to **Run mode**, where you can test how the stack runs.

At the bottom of the Tools Palette (you may have to click on the small arrow on the right to open it) are the drawing and paint tools, including those for creating **vector drawings** (at the top) and those for creating **paint (bitmap) images** at the bottom. Vector drawings are geometric, based on mathematical formulae. They are employed by drawing programs like Adobe Illustrator and take up much less memory than paint images. Vector drawings are relatively simple.

Paint (bitmap) images, used by programs like Adobe Photoshop, are generally used for relatively detailed images, such as photographs, with more colors than in a vector illustration.

The best ways to move a control (object) from the Tools Palette to a card:

- Double click on the control's icon (in Edit mode). This places the control at the center of the card.
- or hold the mouse down over the icon and drag the icon to wherever you want on the card.

Every object in LiveCode, including the stacks and cards, has a **Property Inspector** and a **Script Editor**. In the **Property Inspector** window you can configure the properties of the object, such as its visibility, color, size and position, text, and many other features. In the **Script Editor** you can write a script that performs an action when you interact with the object, commonly by left-mouse-clicking on it. Any object, including stacks, cards, buttons, fields, scrollbars, movies, bitmap paint images, and vector drawings, can have its own script.

The script editor of any object can be opened in a number of ways. Either:

- Right click (Control-click for single button mouse) on the object and select **Edit Script** (or **Edit Card Script** or **Edit Stack Script**, if those are your interest). Try this with the card you see. Personally, I find right-clicking the easiest way to open a script.
- or choose **Edit Script** from within the Property Inspector by clicking on the little arrow at the top of the Property Inspector window (**Fig. 3-1**). This reveals a pulldown menu with the option of **Edit Script**, among other things.

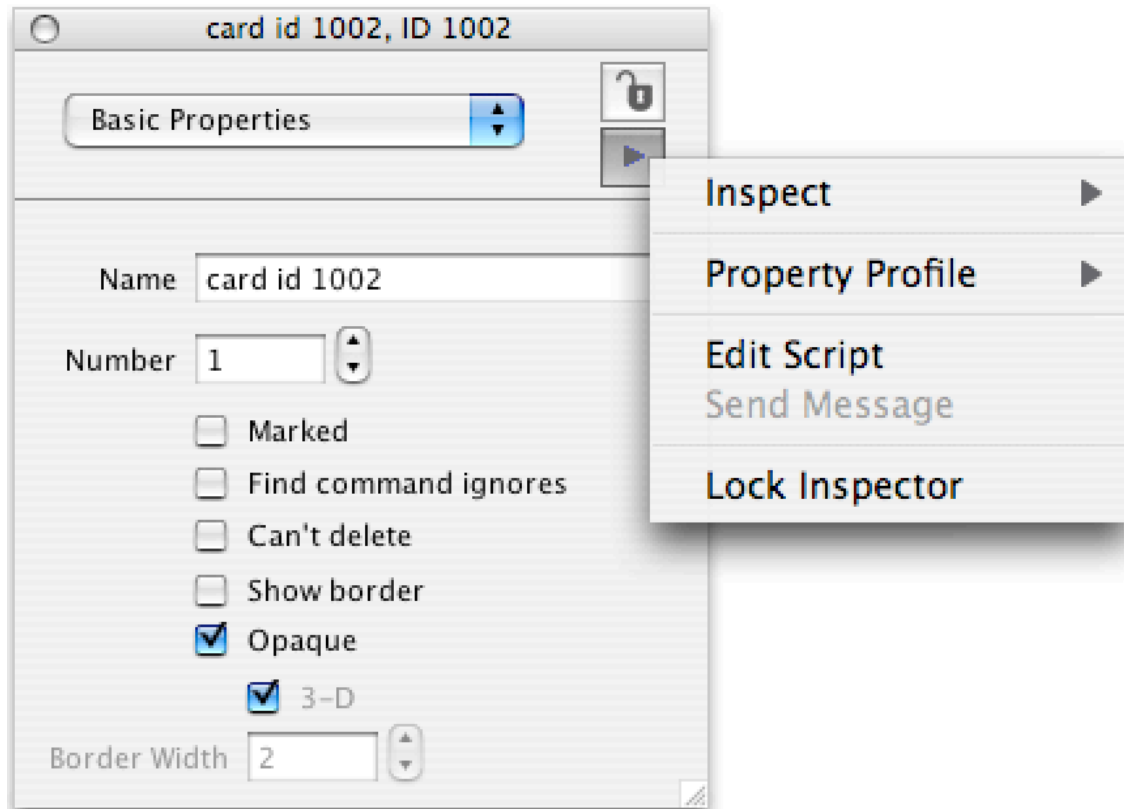


Fig. 3.1. Card Property Inspector

- or choose **Object/ Object Script** (or **Card Script** or **Stack Script**) from LiveCode's **Menu Bar** (Fig. 2-2).
- or click on the **Code**(Script) icon in the icon menubar (Fig. 2-2).
- or press Command-E (Macintosh) or Control-E (Windows) to open a control. (The "E" is for "Edit".)
- or click on the object while holding down the option and command keys (Mac)(control/alt on Windows).

You don't have to do all of these. Just use the most comfortable method for you.

Here we will write a script to navigate from the mainstack to the substack:

1. With MyMainstack in view, double-click on the Tools Palette's **Rectangular Button** icon to place a rectangular button in the center of stack MyMainstack.
2. Right click on the Button and select its Property Inspector.

3. **Name** the button **GoMySubstack**. (Make sure you are working with the button property inspector, not the stack property inspector.)

4. In the **Label** field of the Property Inspector, type **Go To My Substack**. The difference between a button's Name and its Label is that Name is the hidden term that is used in the button's script, while the Label is what the user sees on the button. If you don't fill out the Label field, the button will use the Name as the label by default.

5. Close the Property Inspector and resize the button so that all of its label is seen.

6. Right click on the button and select Edit Script from its menu. This opens the script editor for the button. A button's script editor by default contains lines that read *on mouseUp* and *end mouseUp*. *On mouseup* signifies the beginning of a "handler", something that "handles" in this case the message *mouseUp* (which is sent when a user clicks on the button), while *end mouseUp* indicates the end of the handler. Our script will consist of these two lines and the script lines we will add between them. Note: a script may consist of many handlers, e.g. *on mouseUp*; *on mouseDown*.

7. Type *go to stack "MySubstack"* between the two lines. This tells the program to open and go to stack MySubstack. Thus, the button script reads:

```
on mouseUp  
  go to stack "MySubstack"  
end mouseUp
```

8. Exit the Script Editor by clicking on the Script Editor's close box. Click "Yes" when prompted if you want to keep the change in the script.

Let's test the script:

1. Select the Run Tool in the Tools Palette.

2. Click on button "Go To MySubstack". The program will open card 1 (green) of stack MySubstack.

Now we will create a button to return to stack MyMainstack:

1. With the Edit Tool selected, click on the first card (green) of MySubStack to make it the topmost and active stack. Double click on the Rectangle button in the Tools Palette to place the Rectangle button in the center of the green card.

2. Name the button GoToMyMainstack, with the label "Go To My Main Stack". Close the Property Inspector.

3. Enlarge the button to see the entire label.

4. Edit the button's script to read:

```
on mouseUp  
  go to stack "MyMainstack"  
end mouseUp
```

5. After applying the script and closing the Script Editor, select the Run Tool in the Tools Palette and click on button "Go To My Main stack". This will navigate to stack MyMainstack.

6. Save your work for future reference. Quit LiveCode.

If you already knew these things, please accept my apologies for walking you through all these steps.

Buttons

Open LiveCode and create a new mainstack. Examine the Tools Palette by placing the cursor over each button type (**Fig. 2-2**). The tooltip will indicate each button's type.

Push button: A rounded button in Macintosh, square on Windows.

Default button: A button that will act without using a mouse, simply by pressing the keyboard Return Key.

Rectangle button: Shaped like a rectangle in both Macintosh and Windows.

Check box button: Can be checked or unchecked. If there are multiple check buttons on a card, one can check off any number of them in combination.

Radio button: Unlike the check button, the idea behind radio buttons is that only one button at a time can be highlighted (its circle filled in). If you highlight one button, the others in the **group** become unhighlighted. These will be discussed further in **Chapter 4** on Groups.

Fields

Label Field: A single-line field designed for placing labels on the screen.

Text Entry Field: A simple text field with no scroll bars. It can have multiple lines.

Scrolling Field: Like the simple Text Entry Field, you can type text in this field, in fact many lines, since a scrolling field has a scroll bar.

Scrolling List Field: This field does not allow the user to type in text. Rather, the programmer preplaces text in the field via the field's Property Inspector (discussed later). When the user clicks on a given line of text, an action is performed, as determined by the field's script. List fields can be made with or without a scrollbar.

Basic Table Field: For inserting items in a table format.

Data Grid: This field object can be used for presenting data in complex ways and will not be discussed in this book.

Menu Objects (Fig. 3-2)

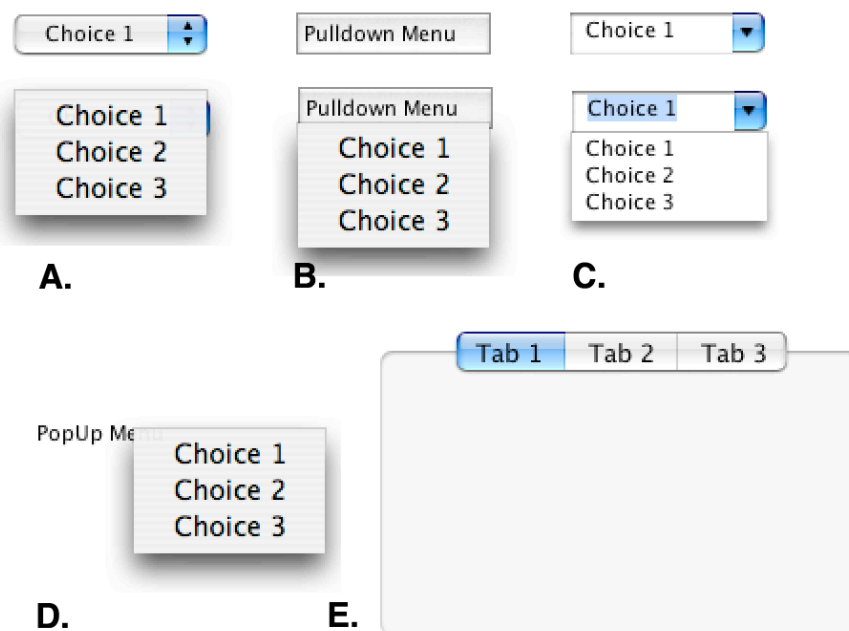


Fig. 3-2. Menu Objects.

While you might think menus would be classified as fields because of their text, they are in fact modified buttons, and in scripts they are referred to as buttons. With the Run Tool active, try out the various menu objects to see what they do.

A. Option Menu: Once the button is set up, the user of the program can select an option from your list. When an option is chosen, the name of the selected option remains visible when the user releases the mouse.

B. Pulldown Menu: The user selects an option from a list that is centered right under the button. Unlike the option menu, the name of the selected option is not visible when the user releases the mouse. This menu is particularly useful for creating your own menu clusters (like File, Edit, View, etc.).

C. Combo Box: Behaves like the Option menu, but the user can also type words into the menu title.

D. Pop-Up Menu: Like the pulldown menu, the pop-up menu has a visible name that never changes, regardless of which choice the user makes. Unlike all the other menu types, the pop-up menu has a transparent background, so if it is not given a name, it could sit unnoticed on the card until clicked on, or it could be given an icon rather than a name.

E. Tab Panel: This menu is designed as a series of Tabs. Clicking on any of the Tabs can elicit a different action.

Scrollbars (fig. 3-3)

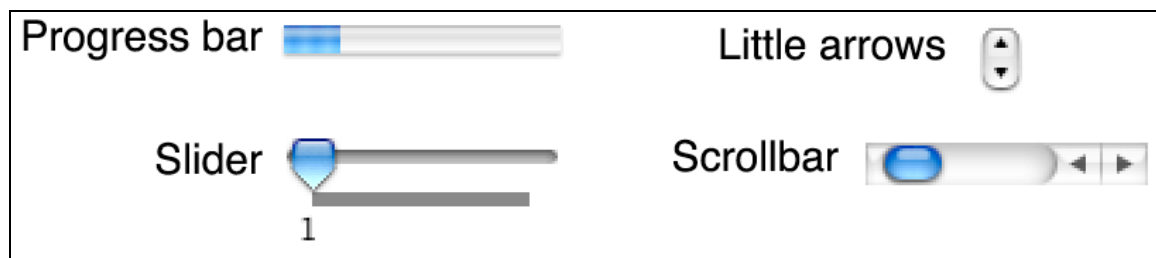


Fig. 3-3. Scrollbars

While you might not think all the following are really scrollbars, that is what the title bars of their Property Inspectors say, and it is how they are referred to in scripts (**Chapter 25**).

Progress Bar: Can be programmed to display the progress of an event.

Slider: Can manually change the parameters of an event (e.g., sound level).

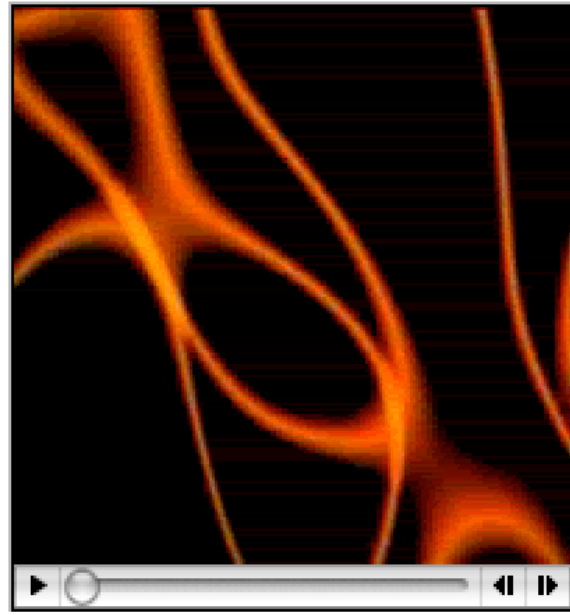
Little Arrows: Can be used to create a customized vertical scrollbar, often used to increase or decrease a numerical count in a field.

Scrollbar: Can be used to create a customized horizontal or vertical scrollbar.

IMAGE AND QUICKTIME CONTROLS (Fig. 3-4)



IMAGE AREA



QUICKTIME PLAYER

Fig. 3-4.

Image Area: Used to import images, particularly in JPEG, PNG, GIF, or BMP format. PNG images can be imported with alpha channels. This means that transparent areas will show up as transparent in LiveCode. JPEG is a compressed format that saves space and is useful particularly for photographs. GIF images do not support as many colors as do JPEGs, but take up less memory and are excellent for cartoon images. GIFs can also be imported as GIF animations and can have transparent areas.

Quicktime Player: Can import Quicktime movies as well as sounds in the common AIF and WAV formats, and image files (including GIF animations). It can even import PDF files.

Draw and Paint Tools (Fig. 2-2)

Vector draw tools: Draw vectors (graphics), which are geometric primitives, such as points, lines, curves and shapes, are based on mathematical equations to represent images, as in a drawing program like Adobe Illustrator. Experiment with the various tools, including the gradient feature in the object's Inspector. While vector drawings are generally less complex than bitmap images, they can be increased in size without becoming pixelated.

Bitmap paint tools: Can paint bitmaps (images drawn as pixels), as in a paint program like Adobe Photoshop. You can use these tools to modify an image that has been imported into LiveCode. You can also use them to create an image

from scratch, in which case using one of the paint tools automatically creates an Image Area control that fills the card. You draw within the Image Area, similar to drawing on a canvas. Bitmap images can be more complex than vector images and are good for photos. Unlike vector images, though, they can become pixelated on attempting to enlarge the original image; you have to plan the size and resolution of the original image in advance of placing it. Experiment with the various tools. There are other controls not shown in the Tools Palette that can be selected through the **OBJECT/ NEW CONTROL** menu.

Quit LiveCode, no need to save (Yay!).

CHAPTER 4. GROUPS

One or more controls on a card can be combined into a group, as follows:

CREATING A RADIO BUTTON GROUP:

1. Create a new mainstack.
2. Place two radio buttons on the card, one aside the other.
3. In Edit Mode, select both of them at once. Objects can be selected as a group by clicking on them in sequence with the Shift key down, or simply by drawing a marquee around them. Each radio button should have its own individual handlebars (**Fig. 4-1**).

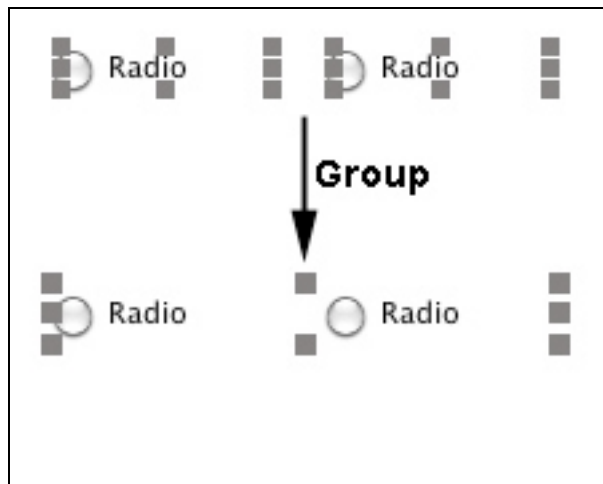


Fig. 4-1. Grouping.

4. Click on "Group" in the Icon toolbar (or choose **OBJECT/GROUP SELECTED**). The radio buttons are now grouped together as shown by a single box around the two buttons, rather than separate boxes around each (**Fig. 4-1**).

5. In Run Mode, click each button. Note that, when grouped, when one radio button is hilited, the other becomes unhilited, which is the expected behavior of radio buttons. LiveCode does this automatically when you combine radio buttons into a group.

CREATING A NAVIGATION BUTTON GROUP:

1. In Edit mode, remove the radio buttons from the card.
2. Place two rectangle buttons on the card. Position them at the bottom of the card as in **Fig. 4-2**.

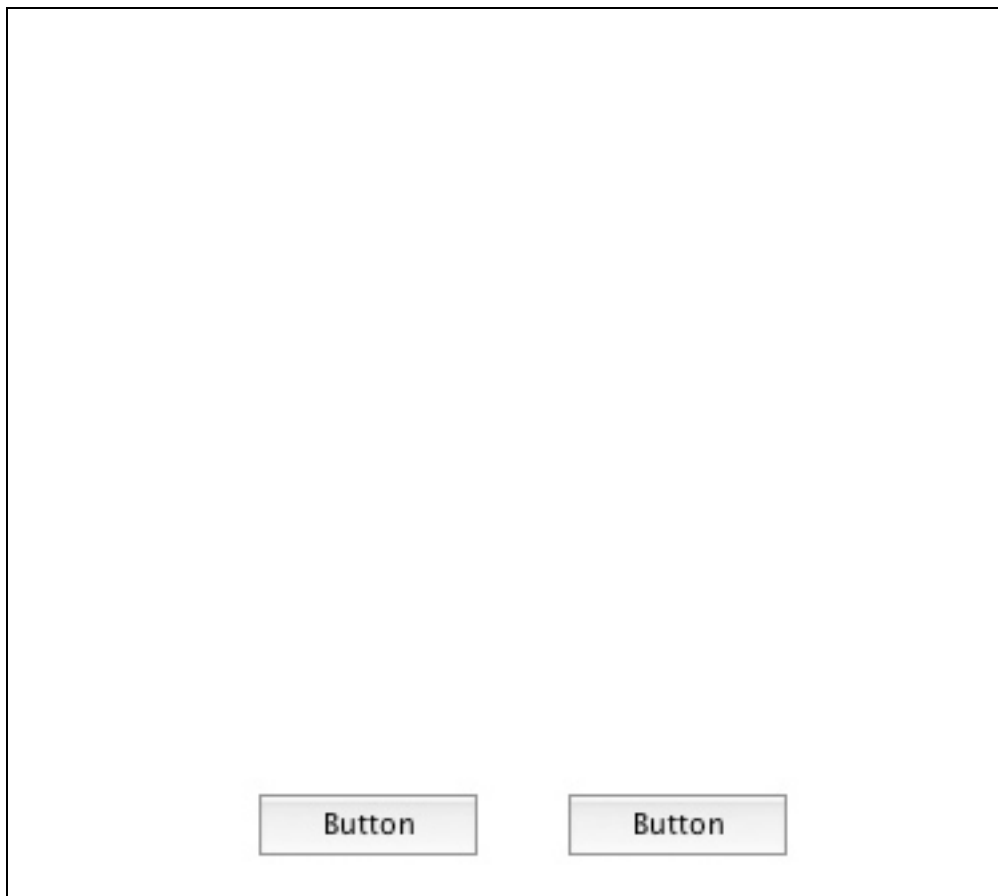


Fig. 4-2.

3. Open the Property Inspector of the Left button.
4. Name the Left button **Left** in the button's Property Inspector's **Name** field (**Fig. 4-3**). Uncheck the **showName** box in the Basic Properties part of the **Left** button's Property Inspector (**Fig. 4-3**) so that no name appears on the Left button.

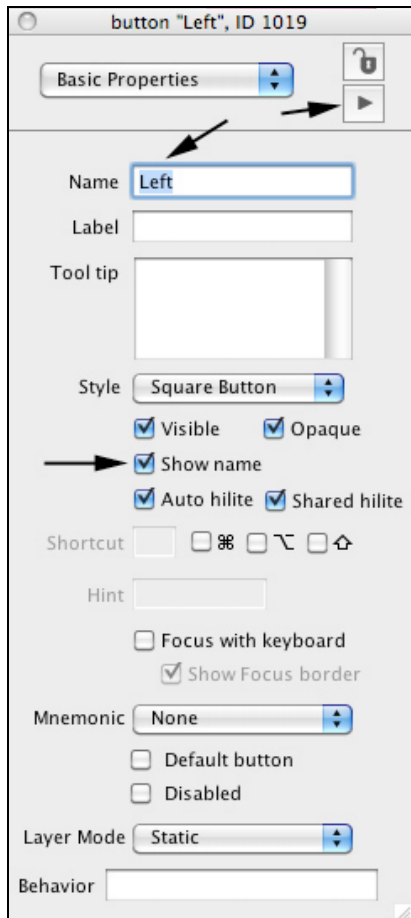


Fig. 4-3.

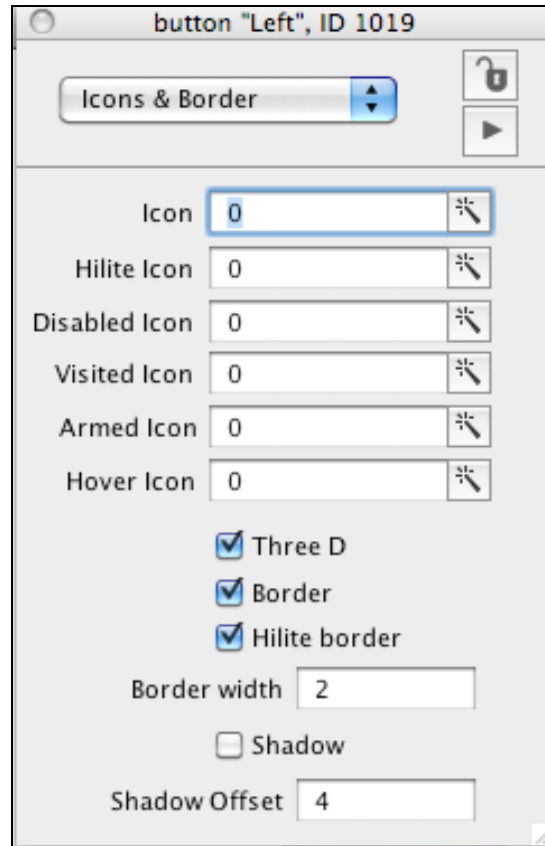


Fig. 4-4. Icons and Border.

5. Select **ICONS & BORDER** from the button Inspector's pulldown menu. (**Fig. 4-4**)
6. Click on the **ICON** magic wand (on the right) to open the list of icons that can be used.
7. Select a left-pointing arrow symbol (**Fig. 4-5**). The "Left" arrow icon will then be visible on the **Left** button.

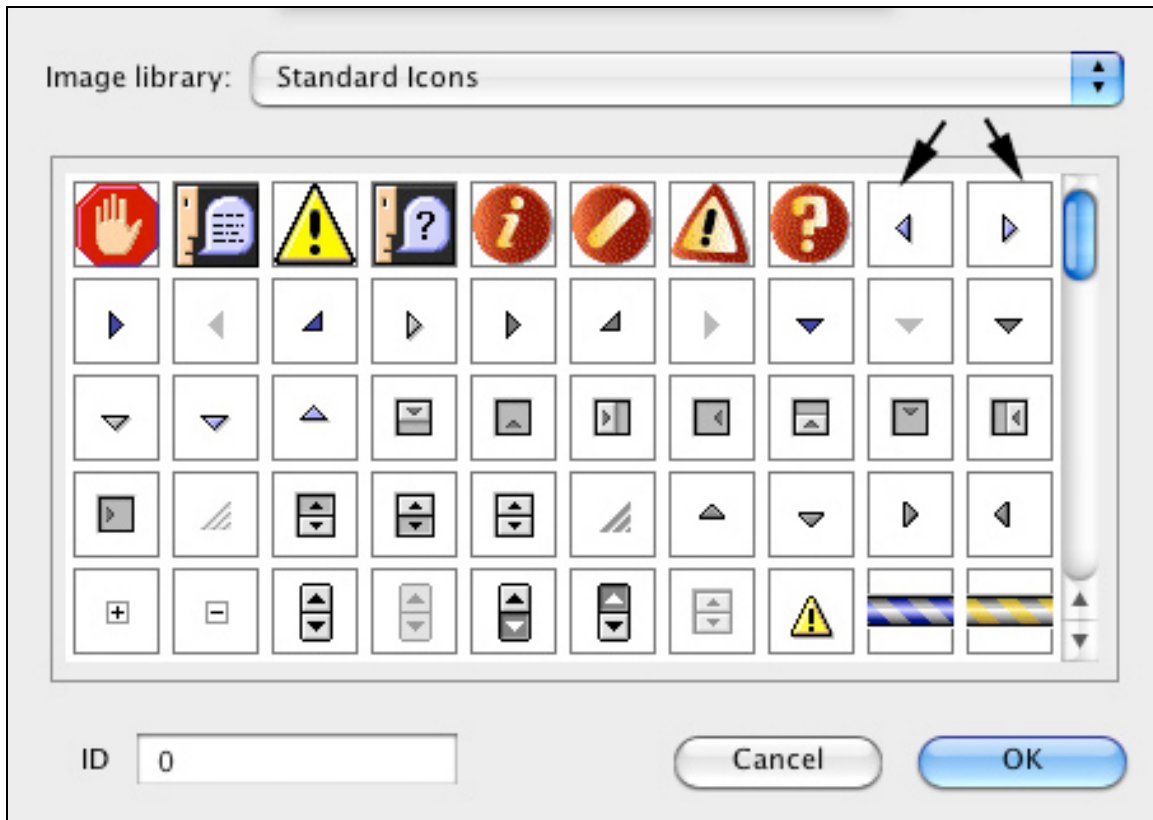


Fig. 4-5. Standard Icons.

8. Open the left button's script editor by clicking on the little arrow at the top right of the button Inspector (**Fig. 4-3**) and selecting **Edit Script** from the pulldown menu that appears. This will bring you into the button's script editor (**Fig. 4-6**). The script editor already contains the words *on mouseUp* and *end mouseUp* to indicate the usual conditions for the script to be enacted (when the mouse is Up) and ended (*end mouseUp*).

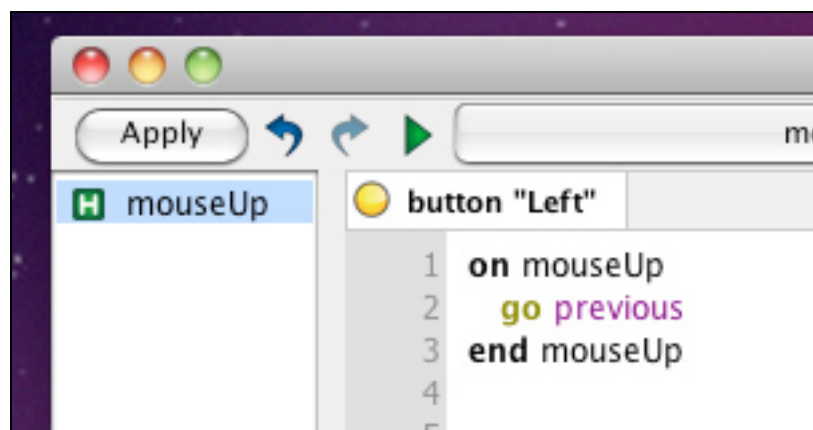


Fig. 4-6.

9. Edit the script to read:

```
on mouseUp
  go previous
end mouseUp
```

and apply the script.

10. Name the other button **Right**, uncheck its showName box, and select a right-pointing arrow symbol. Have the script of the Right button read:

```
on mouseUp
  go next
end mouseUp
```

Save your work as “Navigation LiveCode”.

Now we will transform the buttons into a group:

1. First be sure that **Select Grouped** is unhilited (non-bold text) in the Icon Tool Bar, showing the **Select Grouped** icon with widely separated corner dots. The reason will be explained shortly.
2. Drag the cursor to draw a marquis around the two buttons and select **Group** from the Icon. You could also use Command-G (Mac) or Control-G (Windows) to do the grouping. You have just created a group Toolbar (**Fig. 4-7**). Note that button handlebars, rather than surrounding each button individually, now surround the two button as one group. The buttons can now be moved and positioned as if they were a single object.

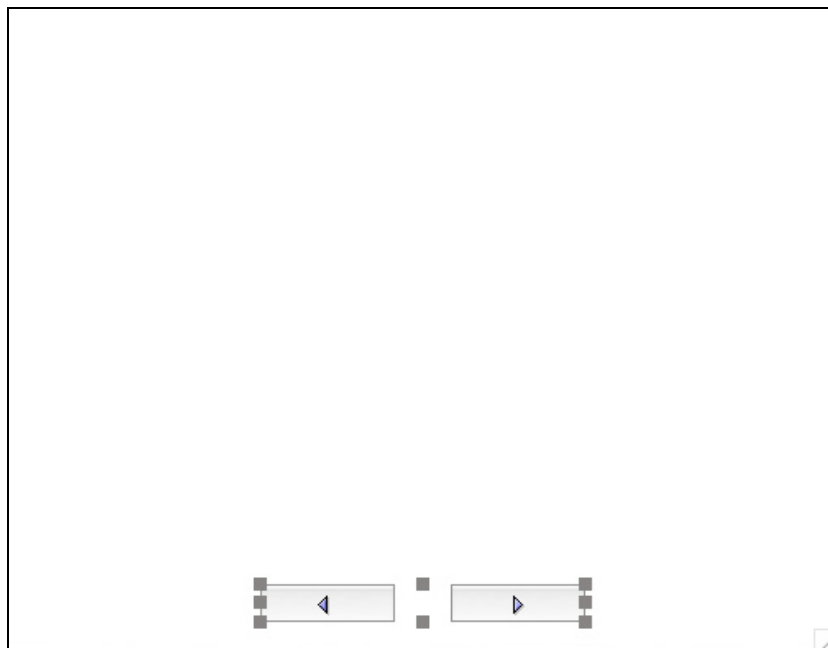


Fig. 4-7.

3. Open the group's Property Inspector by clicking on the group's Inspector icon in the Icon Toolbar, and name the group **Navigation**. (Groups have their own Property Inspectors and Script Editor.)

4. Place a text entry field in the center of the card, simply to identify the card.

5. Create a new card (**Object/New Card**). Note that this card will be blank.

6. While on this second card, select from the Menu Bar **Object/Place Group/Navigation**. This will place the group on the second card, too.

In Run Mode, note that clicking on either the left or right arrows will move you from one card to the other, moving either back or forward.

No matter where you place the group on one card, it will appear in the same place on the other card.

Say you want to make many cards now, all with the same Navigation group. You don't have to go through the tedium of creating blank cards and then placing the Navigation group on each. This can be done automatically, as follows:

1. Open the Navigation group's Property Inspector. Be sure that the box labeled Behave Like a BackGround (**Fig. 4-8**) is checked. Now create a new card (**Object/New Card**). The new card will automatically contain the **Navigation** group.

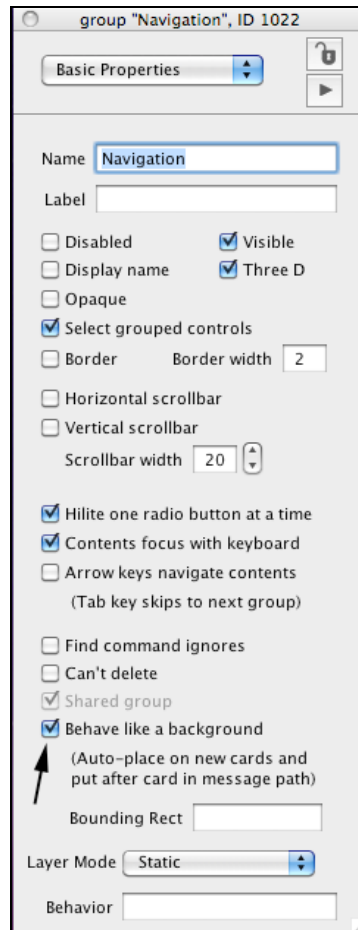


Fig. 4-8. Behave Like a Background.

Now you have the navigation buttons on all three cards. A most valuable feature of putting a group on different cards is that a change to the group on one card immediately appears on all the cards. For instance, try moving around the Navigation group on one card. The position change will appear on the other 2 cards as well, keeping the buttons consistently positioned.

You can also group a single object with itself. In that case, its bounding box appears somewhat larger than in the ungrouped state. Such grouping can be useful if you want the object to appear on a number of cards, always in the same position.

A group is an object in itself, with its own Group Property Inspector and Group Script Editor. Moving a group moves all the controls within it. Deleting a group on one card deletes the group on all the cards in which the group appears.

But what if you want to remove the group from one of the cards but not from the other cards? To do this, select the group on the card in which you want to remove the group. Select from the Menu Bar **Object/Remove Group**. The group will selectively disappear on that card. If you had instead pressed the **Delete**

button, LiveCode will warn you “The group is placed on multiple cards, really delete it?” to let you know that pressing the Delete button would remove the group from all cards on which it appears.

The Select Grouped Feature:

The icon in the Icon Toolbar titled **Select Grouped** can be a source of confusion, but shouldn't be. Let's examine what this does:

When a group, in Edit mode, is selected, showing its bounding box around the group as a whole, checking **Edit/ Select Grouped Controls** changes the group's appearance, so that now each of the controls within the group has its own individual selection box when you click on any of the controls (so you can then “Select” each control within the group individually in Edit mode). (The 4 dots around the **Select Grouped** icon also move inward, confirming that each control within the book will have its own selection box.)

This does not mean that the group has been ungrouped. The group is still there. It is simply a way in which the programmer can conveniently make individual changes to the properties of any individual control within the group. You can, for instance, alter the size or position of one of the group's controls, or modify any of the control's other properties or its script. Those changes will take place on all the cards that contain the group. Some programmers may choose to always leave the group in **Select Grouped** mode (compressed dots) for the convenience of quickly modifying components of the group when desired.

If you try to **delete** one of the controls in **Select Grouped** mode, that same control will be deleted on any of the cards that contain the group. But you cannot **add** a new control (e.g. another button or field) to the group when **Select Grouped** is hilited (compressed dots). In order to do that, **Select Grouped** needs to first be unhilited (dots far apart). Let's do that:

1. Go to the first card (the one with the field on it).
2. Click on and unhilite the **Select Group** icon in the icon Toolbar (so that the dots on the icon's corners are far apart).
3. In Edit mode, click on the Navigation button. The button's dot handles should then surround the group as a whole.
4. Select the **Edit Group** icon in the icon Toolbar. Note that the field you had placed on the card becomes invisible, enabling you to focus solely on editing the Navigation group! In this mode you could remove a control, add a control, modify the size or position of a control, etc. When you are finished editing a group, click again on the **Edit Group** icon; this returns the card to the normal state where you

can see the rest of the contents on the card, the field in this case. The changes you have made will occur in all the cards in the stack that contain the group.

If you really wanted to ungroup a group, click on the **Ungroup** icon in the Icon Toolbar.

To review:

When **Edit Group** is chosen, all the objects on the cards, other than the group that is to be edited, are hidden, enabling the programmer to focus attention solely on the group to be edited. In this mode, you can add other controls to the group, as well as make any other changes to the group. So **Edit group** provides more flexibility than does **Select Grouped** in editing the group.

So why not just use **Edit group** to do all the editing, since it is the most versatile? The only problem with **Edit group** is that when you use it on a group, all the other objects on the card (the field in this case) are hidden, so you cannot see them for reference as you modify the positions of controls within the group itself. So use **Select Grouped** (the dot handles surround each control in the group without removing from view other controls on the card) for all the modifications you want to make to individual controls in a group, except if you want to add or delete a new control in the group, in which case you would need to use **Edit Group**.

When you finish editing a group using **Edit Group**, either click again on the **Edit Group icon** in the Icon Toolbar, or choose **Object/ Stop Editing Group** from the top LiveCode menu bar. Then, you have stopped editing, and all the controls on the card will be visible again.

A group can contain a scrollbar, so that buttons, fields, or images in the group will scroll with the group as whole!

Quit LiveCode. There is no need to save your work (but you can if you really want to).

CHAPTER 5. THE APPLICATION BROWSER

Open **MyTutorial.livecode** and open its **Application Browser (Fig. 2-5)** by selecting **Tools/ Application Browser** from the LiveCode menu bar. This important tool lists all your stacks (main and sub), their cards and other objects on the cards. It also lists any **audioClips** and **videoClips** that you have directly imported. Since audioClips and videoClips remain unseen unless they are referred to in the scripting, the application Browser lets you know they are there and reminds you of their name so they can be referred to in a script. You can

also check how an audioclip sounds, or a movie looks, by double-clicking on its name in the Application Browser.

If you do not directly import an audioClip or videoClip and incorporate it as part of the stack, but have simply referred to it externally via a Quicktime Player object on the card, the referenced sound or movie will be referenced within the Player object, and not listed in the audioClips or videoClips sections.

Right clicking (or Control-clicking with a one-button mouse) on any card listed in the Application Browser brings up a menu that allows you to quickly go to that card or bring up the card's Property Inspector or Script, which you can change. You can, for instance, change the order of the card within the stack by changing the card's number within its Property Inspector.

Right-clicking on any of the column headings of the Application Browser brings up other options for column heads. Just passing the cursor over a column head reveals the column's purpose.

On the right side screen of the Application Browser (you may have to expand the Application Browser window to see this), controls on each card are listed (the Green Card, for instance, should have one, a button). The sole card of stack MyMainstack has a GoMySubstack button that can be seen in the right side screen of the Application Browser.

At any given time, there are many other stacks that are working in the background by default as part of the LiveCode system environment. You can see these in the Application Browser by selecting **VIEW/ LIVECODE UI ELEMENTS IN LISTS**. Yes, the whole development environment of LiveCode itself is written in Livecode! However unless you plan on rewriting parts of this interface (not for the faint of heart) it is better to leave this option unselected.

Quit LiveCode. No need to save.

CHAPTER 6. THE MESSAGE FLOW HIERARCHY

When the mouse cursor acts on a button, it sends a variety of messages to the button, including:

mouseDown – When the mouse button is pressed down

mouseUp – When the mouse is released while still over the button

mouseEnter – When the mouse enters the boundaries of the button

mouseLeave – When the mouse has left the button

mouseRelease – When the mouse is released while the cursor is outside the button

mouseMove – When the mouse is moving within the button's boundaries after entering the button

mousestilldown – Actions that occur continuously while the mouse is down

To respond to one of these messages you would typically place a message handler in the button's script. For instance:

on mouseDown -- means "When the mouse is Down, do the following:"

beep-- issue a beep sound

end mouseDown -- indicates that the *mouseDown* directions are over

on mouseUp

go to the next card

end mouseUp

The above *mouseDown* and *mouseUp* instructions can both reside in the button's script at the same time. Together, they are called the button's **script**. Individually, the *mouseDown* and *mouseUp* instructions are called **handlers**. So in this example there is one script with two handlers. If a script has more than one handler with the same name, only the first handler is executed.

If the button (or other control) does not contain any handlers for the sent message, the message, e.g. *mouseUp*, passes right through the control, searching for other underlying objects that may have a *mouseUp* handler. The message searches along a fixed route, going first from the control to any non-background group the button may be in (if there is one), then to the card, and then to any groups that are acting as a background (in order of number), then to the stack (substack if the control is on one, then to the Mainstack), finally to the LiveCode engine, until the message comes to a handler in one of those places, which traps and carries out the particular mouse message (**Fig. 6-1**). For instance, if the card contains a *mouseUp* script with the command *beep*, and the button on the card contains no *mouseUp* handler, clicking on the button will send the *mouseUp* message to the button, but since there was no *mouseUp* handler to trap it, will send it on to the card, where it will be trapped by the card's *mouseUp* handler and generate a beep.

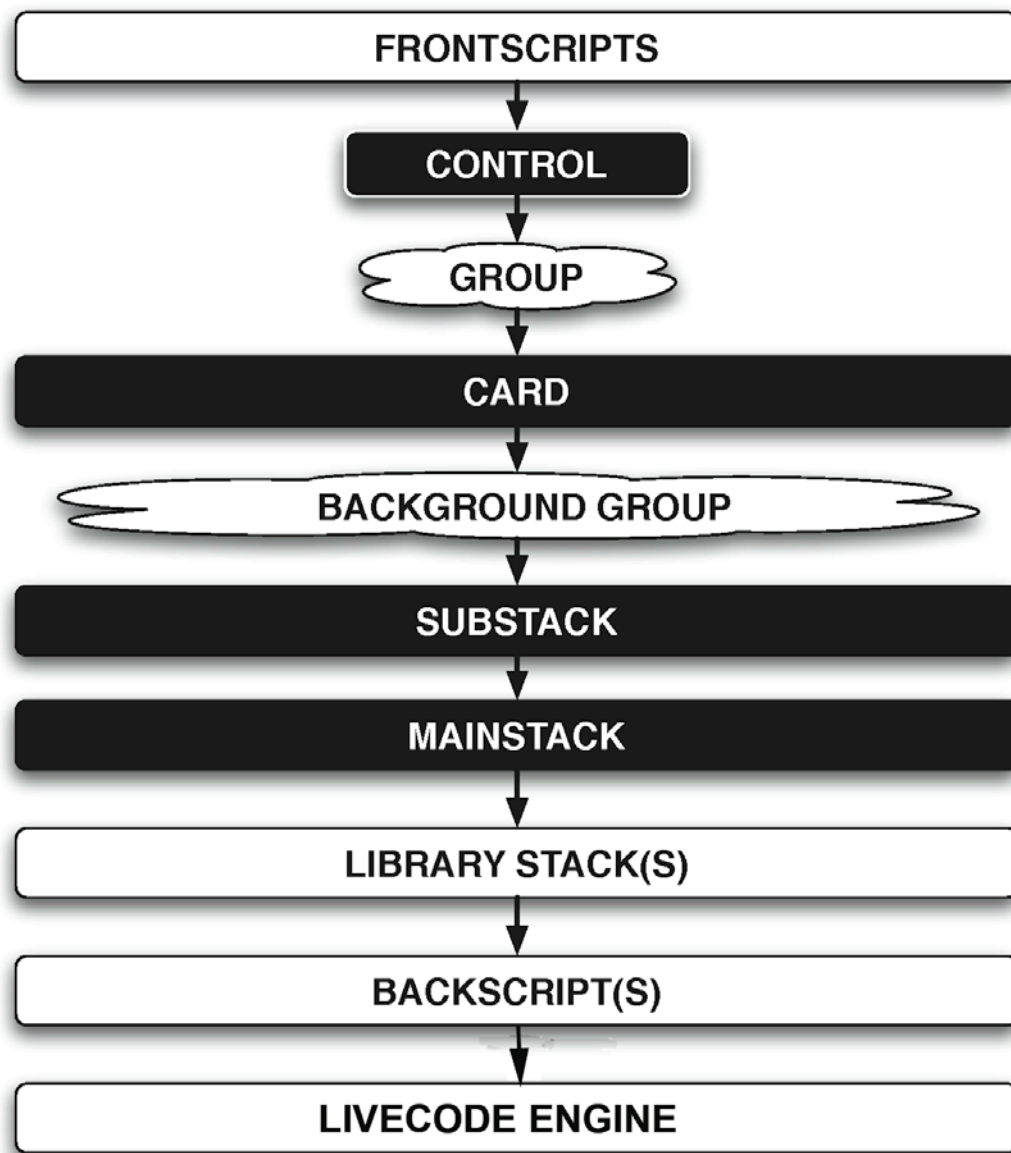


Fig. 6-1. LiveCode Message Hierarchy.

Note in **Fig. 6-1** the 4 areas with dark backgrounds. These generally will be the only points in the message flow that you will need to be concerned about in most cases: the **control**, **card**, **substack**, and **mainstack**, in that sequence. There are, however, other places where you may place handlers:

- **A frontscript** is a special script you might write if you want it to be the first area to receive a command, even when the mouse clicks on an object.

- A non-background **group** is the next waystation after the controls is clicked. If a group's **Behave Like a Background** property is checked, the group (now called a background) is also situated just beyond the card in the chain.
- **Library stacks** are supplementary stacks, whose scripts you may wish to use with the *start using* command.
- A **backscript** is a script that you want to be almost last in the message chain, just before reaching the LiveCode Engine.

As an example, if a control (and non-background group, if present) have no *mouseUp* handler, but the card has a *mouseUp* handler directing to go to the next card, clicking on either the control or the card will result in going to the next card.

Or, if the stack script, rather than the card script, has the *mouseUp* handler, and there are no other *mouseUp* handlers along the route, then clicking on the button will activate the stack's *mouseUp* handler.

If the button simply has the handler:

on mouseUp

end mouseUp

with no instructions as to what to do when the mouse is up, this is still considered a trapping handler (provided you do something minimal in the handler, like typing a space, or clicking **Apply** in the Script box).

There are several ways you can apply a script when leaving the Script Editor:

- Click on the Script Editor's close button to close the Script Editor. You will be prompted to answer whether or not you want to save the script.
- Or, select **Apply** from the Script Editor's **File** menu or click on the script editor's **Apply** button (which saves the script), and then close the Script Editor.
- Or, just press the **Enter** key twice. The first time you press **Enter**, LiveCode applies (saves) the script, but does not close the Script Editor window (you might want to leave the Editor window open when confirming that the script works). The second time you press Enter, the Script Editor window closes.

Where should one put handlers -- in the script of the control, in the script of the card, or in the script of the stack? To illustrate, consider the following two handlers:

```
on mouseUp
  calculateEverything -- (a made-up word)
end mouseUp
```

```
on calculateEverything
  <do this long and detailed calculation>
end calculateEverything
```

You could have both of these handlers in a single button script. Then when the mouse is up, the script will carry out the long and complicated calculation.

However, what if you want to use this script on every card in the stack (e.g., a stack of invoices, in which the long and complicated calculation needs to be carried out on every card. Then it would make more sense to leave the *mouseUp* handler in the button, but place the *on calculateEverything* handler in the stack script. In that way, you don't have to keep duplicating the *on calculateEverything* script on every button or card. Moreover, if you decide to make some changes to the "long and detailed calculation," you don't have to tediously change it in each card's button or card. You just need to change the script once, in the stack script.

Thus, it requires a little judgment as to whether to place scripts in objects, cards, or stack.

Putting a script in a card makes it available to all objects on the card. Putting a script in a stack makes it available to all cards in the stack and their objects. Putting a script in a mainstack makes it available to all the substacks as well.

Scripts Inside Groups – Caution!:

Be cautious in assigning scripts to background groups, particularly those with a *mouseUp* handler. It can lead to confusing results! A background group, however small visually, occupies the entire space behind the card (**Fig. 7-1**). Thus, by clicking on an empty area of the card, one may inadvertently trigger a background group script. An alternative is to not use a background group, but rather to just use **OBJECT/PLACE GROUP** to place groups. When you make a new card, the group will not automatically be placed on the new card, as a background group would, but the placed group will still have the same functionality.

SECTION 2. SCRIPTING

The original HyperCard language had only about 150 scripting words. LiveCode has close to 2000 and continues to expand. Rather than attempting to learn all of

these words at once (many are rarely used) the relatively few key scripting words presented here (about 150) should suffice for the vast majority of your needs.

However, you often may want to consult the excellent LiveCode dictionary not only for words not covered in this book, but for more detailed information about the words described below and related words.

The screenshot shows the LiveCode Dictionary application. On the left is a sidebar with a category list including Library, Browser, Common, Database, Font, Geometry, Internet, Printing, Profile, Speech, SSL, Video, XML, XML-RPC, Zip, Object, Language, Command, Constant, Control Structure, Function, Keyword, Message, Object, Operator, and Property. The main area is divided into two panes. The top pane is a table listing keywords related to 'card'. The bottom pane provides detailed information for the selected keyword, 'card'.

Keyword	Type	Syntax	Platforms	Operating Systems
card	object		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
card	keyword		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
cardIDs	property	get the cardIDs of stack	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
cardNames	property	get the cardNames of {group stack}	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
closeCard	message	closeCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
create card	command	create card {cardName}	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
deleteCard	message	deleteCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
exit to HyperCard	control st	exit to HyperCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
exit to MetaCard	control st	exit to MetaCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
exit to SuperCard	control st	exit to SuperCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
hypercard	keyword		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
metacard	keyword		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
new card	command	new card {cardName}	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
newCard	message	newCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
openCard	message	openCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
preOpenCard	message	preOpenCard	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
printCardBorders	property	set the printCardBorders to {true false}	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS
recentCards	property	get the recentCards {of stack}	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
revSetCardProfile	command	revSetCardProfile {profileName}, {stackName}	Desktop, Server, Web,	Mac OS X, Windows, Linux
show cards	command	show {number all} cards	Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
supercard	keyword		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android
templateCard	keyword		Desktop, Server, Web,	Mac OS X, Windows, Linux, iOS, Android

card
Type: object
Synonyms: cd
See Also: templateCard Keyword
Introduced: 1.0
Platforms: Desktop, Server, Web and Mobile
Supported Operating Systems: Mac OS X, Windows, Linux, iOS, Android
Summary: An object type that is a single page of a stack.
Examples: go to first card
set the marked of this card to true
Use the **card** object type to display different sets of controls in the same stack window.
Comments: A card corresponds to a single page of a stack: one card of each stack can be seen at a time. Each stack contains one or more cards.

LiveCode Dictionary

To access the scripting dictionary, select **Dictionary** from the top LiveCode tools bar. To access all the LiveCode words, be sure the “All” option is selected at the top of the leftmost column.

You have already been exposed to some scripting words. The good news is that you don’t have to remember all of the many useful words. They are remembered for you, being so easily accessible through the tooltip positioned over the various properties in the Property Inspectors. You can avoid much scripting by just

setting the object properties manually through the Property Inspectors (discussed in Section 3).

Professional scriptors often try to write script in the briefest terms with a minimal number of lines of code. Sometimes, though, it is better to write a longer script for clarity, especially if other people are going to read your script.

The Comment Sign

It is frequently desirable to make notations in the script to remind you and others about what your script is trying to accomplish at various points. Scripts can get so complex as to give even their original creator a problem in remembering the reason for what he/she did.

In order to make such comments within the script editor, it is important that LiveCode does not attempt to interpret your notes as an actual script. Thus, wherever there is a double-dashed notation:

--

LiveCode knows that anything that follows that notation on that line should be ignored and is not a script. For example:

```
on mouseUp
beep -- The beep is a simple message than can be used in testing scripts.
end mouseUp
```

In the above script, LiveCode ignores the comment “The beep is a simple message than can be used in testing scripts,” since it follows the double dash. A pound sign (#), or // is also acceptable to signify a forthcoming comment on that line.

If you want to apply a comment to a very long segment of text (or block a long segment of script), this can also be done by placing /* at the beginning and */ at the end of the text sequence:

```
/******
This script was borrowed with permission from the Acme Script Writing
Company, www. acmescript.com.
Is was modified slightly on June 12, 2008
*****/
```

The comment sign can be used to deactivate a number of script lines at once:

```
on mouseUp
--< do A>
<do B>
```



```
--<do C>  
end mouseUp
```

In the above script, the only thing that will be carried out is “B”.

The comment sign can also inactivate an entire handler, simply by putting the double dash right before the first line of the handler. For instance:

```
-- on mouseUp  
  <do something>  
end mouseUp
```

The above script as a whole is inactive, because the dashes were placed before *on mouseUp*. There is then no need to place dashes before the other lines in the script.

Scripting pearl: Sometimes, when reading someone else’s button script, you come across an unfamiliar word and you don’t know whether or not this is an actual LiveCode dictionary word or one made up by the programmer that refers to a handler the developer created somewhere else in a card or stack script. By right-clicking on the word, LiveCode will take you directly to the dictionary if it is a legitimate LiveCode word, or directly to the developer’s handler if it is a word made up by the developer. It is also useful to name the word something that immediately lets the user know that it is made-up, e.g. *MyVariable*.

The terms *on* and *command* are synonymous, but some people prefer to use *command* instead of *on* as a reminder that the word that follows is their own made-up word and not an established LiveCode word

CHAPTER 7. MOUSE-RELATED WORDS

We begin with mouse handlers, since they will be used in scripting examples described below.

on mouseUp
on mouseDown
on mouseEnter
on mouseLeave
on mouseRelease
on mouseMove
on mousestillDown

Note that it is conventional to capitalize the “U” in mouseUp and the “D” in mouseDown, etc., for easier reading. However, LiveCode is generally case-insensitive and it makes little difference whether you capitalize letters (Speech command voice names are an exception – see **Chapter 20**).

Note also that a string (a sequence of characters or words, e.g., the words, “The mouse has just been pressed”), as in the script:

```
On mouseUp
  put “The mouse has just been pressed” into Message Box
End mouseUp
```

has to be in quotes in the script or LiveCode won’t understand. Actually, if there were only one word in the quoted text, quotes are usually not needed, but it is good practice when referring to a text string, regardless of whether it is one word or more, to place in quotes any object name (e.g., the name of a stack, card, or control). This will help distinguish such words from LiveCode dictionary words and variables (discussed in **Chapter 11**), which are never in quotes.

It is not necessary to use the Script Editor to practice many of the commands in this book. You can write script in the Message Box. For instance, instead of a handler within a button that reads:

```
on mouseUp
  beep
end mouseUp
```

you could just type *beep* in the Message Box (**Tools/Message Box**) and press Return or Enter.

Also, instead of using the mouse to directly click on a button, it is possible to direct LiveCode, through scripting, to click on a button or other object. For instance, you could type in the Message Box:

```
click at the loc of button “MyButton”
or
send “mouseUp” to button “MyButton”
or
dispatch “mouseUp” to btn “MyButton”
```

all of which activate the mouseUp script of the button.

In the Message Box, pressing the up or down keyboard arrow scrolls through past scripts that were entered in the Message Box, so you don’t have to retype them. This can help when you want to retry a script a number of times.

Other mouse-related words:

the mouseH
the mouseV
the mouseLoc

The mouseH -- the horizontal distance of the cursor's hot spot from the left side of the card.

the mouseV -- the vertical distance of the cursor's hot spot from the top side of the card.

If the *mouseH* is 100, for example, and the *mouseV* is 150, then the (cursor) is at coordinate position (100,150), which is the *mouseLoc*, namely (*mouseH*, *mouseV*).

the mouse

Just saying *the mouse* tells LiveCode to let you know the state of the mouse, whether up or down.

the mouseClick

The mouseClick tells you whether or not the mouse has been clicked (*the mouseClick* is *true*)

Examples:

on mouseUp

wait until the mouseClick -- i.e., don't do anything until the mouse has been clicked.

beep

end mouseUp

In the above script, you can wait as long as you want, but the beep won't come until you click the mouse somewhere on the card.

CHAPTER 8. NAVIGATION COMMANDS

go

The *go* command tells LiveCode to go somewhere.

Go next means to go to the next card in the stack.

Examples of equivalent scripts:

go to the next card of this stack
go to the next card
go to next card
go next card
go next cd
go next

LiveCode assumes all of the above scripts mean “go to the next card of this stack”. Thus, if you are on card 2 of a 5-card stack, *go next* will take you to card 3.

Note that the script word *the* is optional here and used just to make the script more English-like. You can also eliminate the word *to* for navigation. *Card* can be abbreviated *cd*.

Other examples:

go previous (or *go prev*) – takes you to the previous card in the stack. If you are on card 2, this command will take you to card 1.

go to card 5 of this stack – You can identify a card by its number, in this case card number 5 in the stack.

go to card “menu” – a card can be identified not only by number but by its name.

go to card id 1006 – A card can also be identified by its unique ID number.

go to stack “MySubstack” -- You can navigate between stacks.

go to card 3 of stack “MySubstack” – You can navigate to a specific card in another stack.

In general, when navigating to a card, it is better to identify it by name, rather than by card number or ID number, because identifying the card by name helps you to clearly identify the card when you review a script. Also, if you refer to a card by its number and then add or subtract cards from a stack, or change their order, the number of that card may change and be inappropriately referred to in the script.

Go back – takes you back to the card you were just on. So if you had jumped from card 1 to card 5 and then issued the command *go back*, this will take you back to card 1 (rather than card 4, which would be the *go prev* command).

Navigation is such an important activity in editing that it is very helpful to remember its keyboard equivalents:

- Command-1 (Mac) or Control-1 (Win) goes to the first card
- Command-2 (Mac) or Control-2 (Win) goes to the previous card
- Command-3 (Mac) or Control-3 (Win) goes to the next card
- Command-4 (Mac) or Control-4 (Win) goes to the last card

push card/pop card

Say you have a script that brings you to another card, which in turn connects you to another and another, etc., and after the user visits all those cards, you want to return to the original card. Issue the command *push card* before leaving the original card. This flags that card as the card of interest to return to. On the last card visited, you would write *pop card* in the script of a Return button. This tells LiveCode to return to the original card that was “pushed”.

For example, a button on a card titled “Invoices” might contain the script:

```
on mouseUp
  push card
  go to card “Authors”
end mouseUp
```

On card “Authors” you might have a button whose script is:

```
On mouseUp
  go to card “Royalties”
end mouseUp
```

on card “Royalties” you might have a button whose script is:

```
on mouseUp
  pop card
end mouseUp
```

The *pop card* is all you need to get back to the original card, “Invoices”, which issued the *push card* command.

CHAPTER 9. GENERAL ACTION COMMANDS

put

Put is the command to put something into a “container”. For instance:

```
put “chocolate” into Message Box
put “chocolate” into message
put “chocolate” into msg
put “chocolate”
```

All of the above do the same thing, namely put the word “chocolate” into the Message Box. Since the *put* command is so common, LiveCode accepts the abbreviated forms as well, including just *put “chocolate”* for putting words into the Message Box.

put “chocolate” into field “mouth” – puts the word “chocolate” into a field titled “mouth”.

put field “mouth” – puts the text of field “mouse” into the Message Box. You could also have written:

put the text of field “mouth”

The above scripts show how flexible LiveCode is in providing a user-friendly English-like scripting environment.

The containers that are the recipients of the *put* commands do not have to be fields or the Message Box. A container can also be a variable, e.g.

put “chocolate” into gMyMouth – a made-up variable word
put gMyMouth into msg

We will discuss variables more fully in **Chapter 11**.

The act of “putting” is not confined to text. One can also use images:

put image 1 into image 2 – This substitutes one image for another.

Bonus script pearl:

*put the name of **this** card* – returns the card’s name in the Message Box
*put the name of **this** stack* – returns the stack’s name in the Message Box
*put the name of **this** button* – returns an error message; the word **this** is used only in relation to a stack or a card, not other objects. For a button you might instead write *put the name of me*. For instance, if the button’s name is “Menu” and the handler in the button reads:

```
on mouseUp
  put the name of me
end mouseUp
```

then, on clicking on the button, the Message Box will read **button “menu”**, the long form of the button’s name, with quotes. If, instead, the script line were *put the short name of me*, the Message Box would just read **menu**, without quotes.

set/ get

Script words can be used to set the properties of any Inspector. You can find these script words using the tooltip positioned over the words within each Property Inspector. If you want to change a particular property, such as the color of a card, you could just click on the card's background color box and choose a color manually. Or, you could do it in a script. E.g.,

set the backgroundColor of this card to "red"

The *set* command, then, is very useful for setting the **properties** of objects.

The *get* command gets some particular information and doesn't do anything with it except store it temporarily in an invisible container called *it*. (*It* is a type of variable, but then again, we haven't discussed variables as yet.) One can then do something with the *it*. For example, say there is a field titled "food" and the field contains the word "chocolate". Then, if one writes the script:

get the text of field "food"
put it into msg

the first of the above two lines will *get* the text "chocolate" from the field titled "food" and put the word "chocolate" into the container called *it*. The second line puts the contents of *it* into the Message Box.

This is the same as saying:

put the text of field "food" into it
put it into msg

More briefly, one could just write: *put field "food" into msg*, or just *put field "food"*. Just different ways of expressing the same thing.

hide/show

You can hide or show stacks, or controls placed on a card. E.g.,

hide this stack
show this stack
hide button "start"
show btn "start"
show fld "info"
show image "rainbow"
hide me – hides whatever object you clicked on
hide menubar – hides the LiveCode menu bar (at the top of the screen)

Note the following common optional abbreviations:

button -- btn
field -- fld
card -- cd
cr-- carriage return, which is the same as return

send

The word *send* (alternatively, *dispatch*) is used to send a message that triggers a handler in a different object. For instance, suppose there is a button titled “MyName” which contains the script:

```
on mouseUp
    beep
end mouse
```

and there is another button titled “Transmitter” that has the script:

```
on mouseUp
    send “mouseUp” to btn “MyName”-- or dispatch “mouseUp” to btn “MyName”
end mouseUp
```

If you click on button “Transmitter” it will send a *mouseUp* message to button “MyName” and there will be a beep.

quit

Simply writing *quit* suffices to close the entire stack file (or standalone, after the standalone has been created). For example:

```
on mouseUp
    quit
end mouseUp
```

Simple? Yes!

answer vs ask

The *answer* command takes the form:

answer <question> with <reply1> or <reply2> or <reply3> or <reply4> -- Up to 7 replies are allowed. For instance:

answer “What color are your eyes?” with “Brown” or “Blue” or “Green” or “Cancel”

Try writing the above in the Message Box. On pressing Return, an answer dialog appears with those choices (**Fig. 9-1**).



Fig. 9-1. Answer box.

Whatever choice you make, whether “Brown”, “Blue”, “Green”, or “Cancel”, those words go into the variable container called *it*, in which case the script can act on that choice. For instance,

```
on mouseUp
  answer "What color are your eyes?" with "Brown" or "Blue" or "Green" or
  "Cancel"
  if it is "Cancel" then exit mouseUp -- the handler stops and nothing is done
  put it -- puts it into the Message Box if you made a choice of eye color
end mouseUp
```

Note that it is necessary to add the line

```
if it is "Cancel" then exit mouseUp
```

because clicking on “Cancel” places the word “Cancel” into *it*, just as clicking on “Brown”, “Blue”, or “Green” would place those words into *it*, and without that line about exiting, the word “Cancel” would be placed into the Message Box. You didn’t have to use the word “Cancel”. You could have used any other word or group of words, such as “None of your business”.

The *ask* command differs from the *answer* command in that its dialog box contains a field in which the user types a response (**Fig. 9-2**). For instance, put the following script in a button:

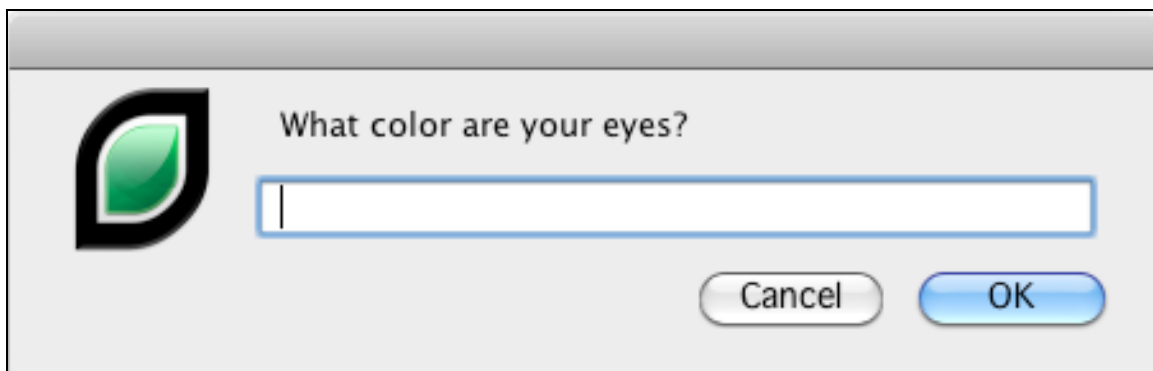


Fig. 9-2. Ask box.

on mouseUp

ask "What color are your eyes?"

if it is empty then exit mouseUp -- Nothing happens if the dialog text box is empty, i.e. the user did not type any text.

put it -- puts what the user typed into the Message Box. Or, you can do something else with it that is more interesting.

end mouseUp

By default, there are just an "OK" button and a "Cancel" button in *ask* dialog boxes. When you click on these, the words "OK" or "Cancel" are not placed into *it*, as would be the case for the *answer* dialog. Rather, the **contents of the ask dialog box text field** are placed into *it* once "OK" is clicked. If "Cancel" is clicked, *ask* dialogs, unlike *answer* dialogs, interpret this as stopping and exiting the script right there.

The *ask* command dialog box field doesn't have to be empty. You can indicate a default text for the field:

on mouseUp

ask "How many beers would you like to order?" with "1"

if it is empty then exit mouseUp -- Nothing will happen if the dialog text box is empty.

put it -- puts "1" into the Message Box.

end mouseUp

In the latter script, the user can order just the 1 beer without having to type in anything, or type in how many to order.

A modification of *ask* is *ask password*, as in:

ask password "What is your security code?"

The above script brings up a dialog box in which the user's typing appears in asterisks (*****) for privacy, and can be used as a password. See the LiveCode dictionary for variations on this.

The *ask* and *answer* dialog boxes can contain icons symbolizing "Error", "Warning", "Information" or "Question" (**Fig. 9-3**), as in the modified script lines:

answer question "What color are your eyes?" with "Brown" or "Blue" or "Green" or "Cancel"

ask question "What color are your eyes?"






	ERROR	WARNING	INFORMATION	QUESTION
WINDOWS				
MAC OS X				
MAC CLASSIC				
LINUX & UNIX				

Fig. 9-3. Ask and Answer icons.

sort

Sort can be used to sort cards or lines in a container, such as a field. For example, imagine you have a stack of cards with a background field titled “Name” at the top of each card. Each card corresponds to a different name. You want to sort the cards in alphabetical order. This is accomplished in the script:

sort cards ascending by field “Name” – sorts in ascending alphabetical order,

or more simply:

sort cards by field “name” – if you don’t specify *ascending* or *descending*. LiveCode assumes *ascending*.

If you wanted to sort the cards in reverse alphabetical order, you can write:

sort cards descending by field “Name”

or numerically:

sort cards numeric by field “zip code” -- sorts ascending numeric by default

sort cards ascending numeric by field “zip code”

sort cards descending numeric by field “zip code”

sort field “MyList” -- sorts the lines in field “MyList” in ascending alphabetical order

wait

In writing several lines of script within a handler, each line represents a different task for LiveCode to carry out as fast as it can. If you want a time delay at some point in the handler, you can use the *wait* command:

```
on mouseUp
  put "Listen to this sound" into field "Listen"
  wait 3 seconds
  beep
end mouseUp
```

The beep will occur after a 3-second delay. You can also portray small time intervals as ticks. A tick is 1/60 of a second. A millisecond is 1/1000 of a second. E.g.,

wait 10 ticks or just *wait 10*

A difficulty with the *wait* command is that no other script will run while the waiting occurs. For instance, suppose you wish there to be a 5-second delay after a user clicks on a button before the command *myCommand* is executed. You could then have in the script of the button:

```
wait 5 seconds
myCommand
```

However, using this construction means that clicking on any other buttons in that 5 second interval will have no effect until those five seconds are up.

A way around the problem is to use the *wait ... with messages* format. E.g., in the button script:

```
on mouseUp
  wait 5 seconds with messages
  mCommand
end mouseUp
```

The user can then effectively click on other buttons, or perform other actions in the 5 second interval.

edit script

Edit script opens up the Script Editor of a stack, card, or object on a card. Examples:

Edit the script of this card
Edit script of this stack
Edit script of btn 1
Edit script of btn 1 of next card

Edit script can be very useful, for instance, if you accidentally created a button “MyButton” with just an *on mouseEnter* handler. For instance:

```
on mouseEnter  
  answer “Why did you do that?”  
end mouseEnter
```

You will then find it difficult to alter the script of button “MyButton” because every time you place the cursor over the button (whether in Run or even in Edit mode), you will frustratingly get that “Why did you do that?” answer box. A way to get into the button script is to type in the Message Box:

Edit script of btn “My Button”

This will open the button’s script editor, where you can make changes with no difficulty.

move/stop moving

The move command moves a stack or control by scripting. Examples:

move this stack from 0,0 to the screenloc – this gradually moves the center of the stack from the upper left portion of the computer screen to the center of the screen.

move btn 1 from “0,0” to “100,125” in 5 seconds – gradually moves button 1 from the 0,0 location on the card (the card’s upper left corner) to 100,125 on the card over a 5-second period.

Setting the *moveSpeed* sets the speed of the move:

```
set the moveSpeed to 10  
move btn 1 from 0,0 to 100,125
```

stop moving btn 1 – stops the movement of the button before the above movement is completed.

You can move a control along a curved path as well. For instance, create a path using the **Freehand Graphic tool**. A graphic object can then move along the path:

move image 1 to the points of graphic 1 in 1 second

beep

The beep sound can be useful as an alarm to alert the user to a significant event. It can also be used during development by temporarily placing the beep command at certain points in a script to see if the script is functioning up to that point. (Another useful word to temporarily put into a script to see if it is functioning up to that point is *put*, as in *put "hi"*. If the script is working up to that point, the Message Box should show the word "hi".

CHAPTER 10. KEYBOARD WORDS

"Keyboard Words" refers to specific keys on the keyboard. Examples:

if the controlKey is down then <perform some action>
if the commandKey is down then <perform some action>
if the optionKey is down then <perform some action>
if the shiftKey is down then <perform some action>

on keyDown pKey is a handler for any particular key that might be pressed. Try putting this into the script editor of a field and then typing within the field any letter or number:

on keyDown MyKey
put MyKey
end keyDown

The Message Box will show each letter or number you type in the field. In the above script, *on keyDown* means "When you press a key down". The word *MyKey* is a container to hold the identity of the particular key you pressed and could be any made-up word. This particular handler puts the name of the letter or number you type into the Message Box. You could do more important things with *on keyDown MyKey*:

on keyDown MyKey
if MyKey is not a number then answer "You must enter a number"
else pass keyDown
end keyDown

The above script uses the conditional if-else format, which is discussed in **Chapter 16**. Briefly, though, the idea is that here you want a field that accepts only numbers, not letters. If the user mistakenly types a letter, the message "You must enter a number" appears, instead of the letter being typed in the field. If it is a number, the *else pass keyDown* part of the script "passes" the number along to the field, where the number then appears.

The *toUpper* and *toLower* words are functions that direct LiveCode to convert any typed letters to upper or lower case. Try this in a field script:

```
on keyDown MyKey
  put toUpper (MyKey ) into the selection
end keyDown
```

This script converts any typed letters into upper case. The reason it works is: When you select text within a field, the selected text is called the *selection*, which is also a type of container. You can *put* things into a container, so if you *put* text *into* a *selection*, it replaces what was selected. If the user clicks in a field but doesn't select any words and there is only an insertion point, this is still a *selection*, but one than consists of 0 characters. If you *put* text *into* that barebones *selection*, the result is text added at that insertion point. In the above *keyDown* script, there is an insertion point in the field just before you type anything. When you *put toUpper (MyKey)* into the *selection*, you are inserting the upper case form of the letter you are typing at the insertion point. This type of scripting is further clarified in the **Chapter 13**, which discusses **Functions**.

The keyboard contains 4 arrow keys: up, down, left, and right. Scripting to direct what happens when an arrow key is pressed has the following format:

```
on arrowKey MyKey
  if MyKey is "right" then beep
  if MyKey is "left" then <do something else>
  if MyKey is "up" then < do another thing>
  if MyKey is "down" then < do yet another thing>
end arrowKey
```

CHAPTER 11. VARIABLES AND CUSTOM PROPERTIES

It

Temporary Variables

Local Variables

Global Variables

Custom Properties

Variables are extremely valuable unseen storage containers into which one can "put" something, particularly words and/or numbers. The purpose of using a variable is to have the script remember some information for future reference. How long the variable remembers its contents depends on the type of variable.

The variable *it* has the shortest term memory. “Local” and “global” variables, and custom properties have progressively longer memory spans.

When naming variables, the first letter of the variable must be either a letter or an underscore and the variable name should not contain spaces. The variable name should not duplicate an established LiveCode script or reserved word and it should not be in quotes.

The way variables are used can be demonstrated in the following examples:

It

As you may recall, in **Chapter 9**, in discussing the *answer* and *ask* dialogs, the response of the user is immediately put into the unseen variable *it*:

```
ask "What is your name"  
put it into field "UserName"
```

When the dialog box appears in the course of the above script, the user types in his/her name and then presses “OK”; the user’s name is immediately placed into the variable *it*. You can then do with *it* whatever you want in the context of the script. *It*, though has a very short term memory. If you have a later spot in the same script that also put something into an *it*, the first *it* is lost from memory, since one can only have one *it* at a time. Thus, if one is going to rely on *it* as a container, it is best to use it immediately.

Any

Any is a quick way to randomly select one of a list of things:

```
put any line of field 1  
put the name of any button of this card -- or on this card  
put any field of this card
```

Temporary Variables

Temporary variables, like *it*, have a short memory. It is good practice to put a small “t” before the name of your temporary variable to remind you that it is only temporary. It can be used only within the confines of one message handler. For instance:

```
on mouseUp  
  put the number of lines in field "data" into tHolder  
  add 5 to tHolder  
  <do some other routine with lots of other scripting>  
  put tHolder into field "Total"
```



```
end mouseUp
```

The message handler remembers what *tHolder* refers to and can act on this information at some further point in the handler's script.

tHolder has a limited memory span, though, since once the handler finishes, *tHolder* forgets what it held. For instance, consider the following two handlers in the same button script (one for *mouseDown* and one for *mouseUp*):

```
on mouseDown
  add 1 to tHolder -- by default, lHolder is originally considered to contain 0
end mouseDown

on mouseUp
  put tHolder
end mouseUp
```

You might expect that the Message Box on *mouseUp* would show a number, referring to the contents of *tHolder*. But the Message Box will only say "tHolder". That is because from one handler to another, the script forgot what *tHolder* meant, so it just puts the word "tHolder" into the Message Box by default.

Local Variables

If one "declares" a local variable (customarily preceded by an "l") at the top of the script, outside the handlers, LiveCode remembers the value for *lHolder* anywhere within the script. Thus, in script:

```
local lHolder

on mouseDown
  add 1 to lHolder
end mouseDown

on mouseUp
  put lHolder
end mouseUp
```

the Message Box would say "1". Not only that, continuing to click on the button will result in the continuous adding of the number. When *lHolder* is declared outside the handlers, the button remembers *lHolder* for the next time the button is clicked. But *lHolder* is not remembered in other buttons or anywhere else in the stack, and the memory totally disappears after the stack is closed.

Like other properties of an object's Property Inspector, scripts are also properties, so you need the *set* command, rather than the *put* command to change a script. For instance, if button "MyButton" has the script:

```
on mouseUp
  put 2 into fld "MyField"
end mouseUp
```

and you wanted to use the Message Box to change this script to:

```
on mouseUp
  put 3 into fld "MyField"
end mouseUp
```

you can't just write:

```
put "3" into word 2 of line 2 of btn "MyButton"
```

This won't work because you can't change a script using the *put* command. You could, however, use a variable in the following sequence:

```
put the script of btn "MyButton" into tHolder – puts the script into a temporary variable
put "3" into word 2 of line 2 of tHolder – modifies the variable
set the script of btn "MyButton" to tHolder – sets the script to the modified variable
```

Global Variables

Even if you declare *local tHolder* in the above example, LiveCode will forget what *tHolder* meant once you are outside that particular object. What if you want LiveCode to remember what a variable means throughout the stack, so long as the stack is open? This can be done with a **global variable**. Customarily, one uses a "g" rather than a "t" at the beginning of the global variable name to remind you that you are dealing with a global memory. It is not necessary to do so, but is considered good scripting practice, as it acts as a visible reminder that the variable is a global. Then the script might look like:

```
on mouseUp
  global gNumber
  put 5 into gNumber
end mouseUp
```

The declaration of the global variable is generally the first line within the handler. If you don't declare it as a global (*global gnumber*), LiveCode assumes *gNumber* is a local variable. When declared as a global, the global variable will be

remembered as long as the stack file is open. If you are, say, on another card (even in another stack or substack), and have a different script handler that wants to invoke the global *gHolder* the distant script handler would read:

```
on mouseUp
  global gNumber
  <do something with gNumber>
end mouse
```

In that case, the original *gNumber* is remembered, even in a distant area of the stack file.

For brevity in scripting, if you have many message handlers within a script and you don't want to declare the global variable at the beginning of each handler, you can just declare it once, outside all the handlers, at the top of the script. Thus a script could read:

```
global gNumber
```

```
on mouseDown
  put 5 into gNumber
  beep
end mouseDown
```

```
on mouseUp
  put gNumber into field "Endresult"
end mouseUp
```

For clarity in scripting, it is wise to give variables names that call to mind what they are used for. For instance, if the variable is supposed to contain a test score, then rather than naming it *gHolder*, it would be more meaningful to name it something like *gTestScore*. A global variable name must start with either a letter or an underscore. Do not give a global variable a name that duplicates that of a Custom Property (see below), since this may confuse LiveCode. Variables should not have quotation marks.

Also, do not use a variable name that begins with *gRev*, since those global variable names are reserved for the LiveCode development environment. The *gRev* variables are always there behind the scenes and do not require declaration for them to be used. For a list of these global variable environment names, click on the **Global Variables** icon in the Message Box, and check the "Show LiveCode UI Variables" box at the bottom.

If you want to declare multiple globals in a script, separate them with commas. E.g.:

```
On mouseUp
  global gfirstGlobal,gsecondglobal,gthirdglobal
  <do something>
end mouseUp
```

Custom Properties

Although global variables have a pretty long term memory, they are remembered only while the stack file is open. Once the stack file is closed, the memory of the global variable is lost. How does one get the stack to permanently remember a variable? This is very simple. For this, we use **Custom Properties**, which have complete long term memory, as follows:

Say there is a substack “MySubstack” which has somewhere in its scripting:

set the myLastScore of this stack to “120”

This declares a custom property, termed *myLastScore* which can be confirmed in the stack’s Property Inspector **Custom Properties** section (**Fig. 11-1**). You will here see the Custom Property *myLastScore* listed, along with its content, which is 120. You could have used almost any word besides *myLastScore*, which is a made-up word. Just don’t use the letters “rev” as part of a variable or a custom property name, since it might be confused with other words used by the LiveCode engine. A custom property should be a single word of which the first character should be either a letter or an underscore (_). It must be preceded by the word *the*, when referred to in a script.

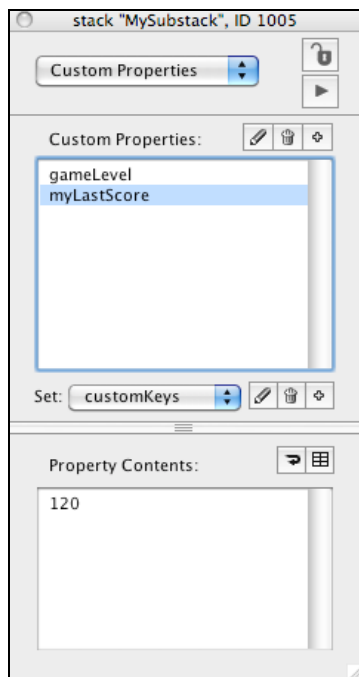


Fig. 11-1. Custom Properties

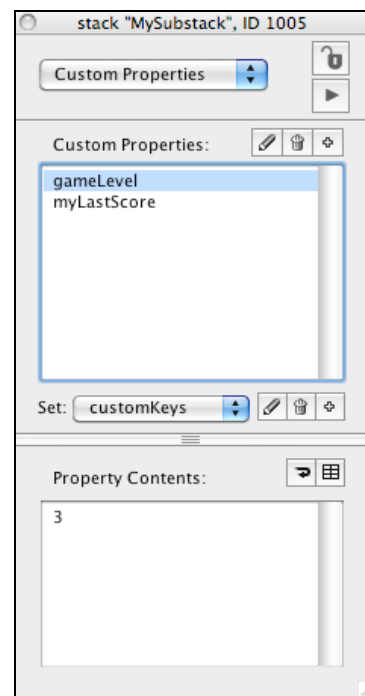


Fig. 11-2. Custom Properties

If there is an additional script:

set the gameLevel of this stack to 3

the **Custom Properties** of *myLastScore* and *gameLevel* are listed in the top field, and their values (click on each of the custom properties to see their values) in the lower field (**Fig. 11-2**). This information is remembered even if you close the stack. You can create as many custom properties as you wish. They don't have to be put into the Main Stack's Property Inspector. They can be placed in a substack's Property Inspector or in the Property Inspectors of any of the objects in the main stack or substacks, a tremendous number of storage rooms!

If you wish, custom property names can be preceded by a "c" (e.g. *cMyLastScore*) to remember that you're talking about a custom property.

IMPORTANT! If you create a **standalone** application, information that a user enters into the standalone **cannot be saved to its mainstack**. Substacks, though, can save information. Therefore, it is better to use a substack as the place for any changes the user might want to make within a standalone. For that reason, many developers simply prepare the mainstack as a single card that connects to its substacks, without putting any features in the mainstack that would require saving on using the standalone.

In addition to placing custom properties in a substack rather than in the mainstack, one needs to do two other things to insure that changes made to a standalone (e.g. typing in a field, creating new values for a custom property) are remembered:

1. The script of the standalone's substack should contain the line *save this stack* prior to closing so that the stack is indeed saved.
2. The programmer should check the box titled **Move substacks into individual stackfiles** in the **FILE/ STANDALONE APPLICATION SETTINGS/ STACKS** section of the LiveCode Menu bar.

The Custom Properties feature is very useful if, say, you want to remember a game level, a quiz score, or any other data after a stack is closed.

In the above example, there are two custom properties, *gameLevel* and *myLastScore*. Together they are part of a set, which by default is called *customKeys*.

To remove all the custom properties from *customKeys*, you can do it either by clicking on the **Custom Keys** garbage can, or you can do it by script:

set the customKeys of this stack to empty

If you want to create another **custom property set** named something else besides “CustomKeys”, e.g. **MyNewSet** you can do this by clicking on the “**New Custom Property set** ”+” icon.

The name of a custom property set should be a single word, of which the first character should be a letter or an underscore (`_`). If you want to add by script a new custom property titled *CorrectAnswers* to **MyNewSet**, do it in this format:

```
set the MyNewSet[CorrectAnswers] of this stack to 45
```

This has added a custom property titled “CorrectAnswers” to the “MyNewSet” custom property set, and also indicates that the number of “CorrectAnswers” is 45.

Although custom properties can be considered a kind of “container”, they really differ somewhat from other containers, such as fields and local and global variables. In the case of fields and local and global variables one can use *put* scripts like:

```
put “Every” before field “Test” -- for a field  
put “6” into tTemporaryLevel -- for a local variable  
put “Total” before gFinalscore -- for a global variable
```

However, you can’t use the *put* command to make an alteration within a custom property. Since custom properties are indeed “properties”, you can only use the *set* command, as is done for all other kinds of properties. Thus:

```
set the gameLevel of this stack to 12 – and don’t forget to include the word the.  
Also, do not use quotes to refer to a custom property.
```

If you want to make more detailed changes to a custom property, you need a somewhat roundabout way, first putting the property into a temporary variable and then setting the custom property to the temporary variable, as in the following:

```
on mouseUp  
  put the myLastScore of this stack into tTempscore -- tTempscore being a local variable  
  put “:Beginner” after tTempscore -- changes the tTempscore variable  
  set the myLastScore of this stack to tTempscore  
end mouseUp
```

Yes, we also used the same technique of using an intermediary temporary variable when we wanted to script a change to an object’s script because scripts and custom properties are properties.

CHAPTER 12. ME vs THE TARGET

Sometimes buttons can do different things even though their scripts are the same, if you use the word *me*:

```
on mouseUp
  put the short name of me after field 1
end mouseUp
```

Then, whenever you click the button, that button's particular name is put after field 1. This saves time in scripting, as every button has the same script. The word *target* can be even more efficient. Consider the following script in a group that contains many buttons:

```
on mouseUp
  put the short name of the target after field "display"
end mouseUp
```

In this case, there are no scripts at all in the buttons in the group; when clicking on a button, the *mouseUp* command passed through to the group, which traps and enacts the message. Since the actual target of the click was the button, the short name of the button is put into the Message Box.

So you can see how the words *me* and *target* differ.

Grab me

Grab me is sort of an oddball command that relies on the word *me*. *Grab me* is used to drag an object in Run Mode when the mouse is down. The button can be dragged all around the card, following the cursor, if it has the script:

```
on mouseDown
  grab me
end mouseDown
```

This can be useful in certain kinds of game development or if you want to provide a degree of user customization in positioning to your stack.

CHAPTER 13. FUNCTIONS

Both a function and a command ask the program to do some action. A function also asks the program to bring back some information, so in a sense it can be

regarded as a special type of command that expects some information to be **returned** first.

TIME AND DATE FUNCTIONS

To illustrate how time and date functions work, consider the following functions as written in a Message Box. Note the necessity of using the word *the* in calling a function, or else using the abbreviated format ():

Put the date -- also written *put date* () or just *the date* – returns, e.g. 3/8/15 if that were today's date, in the Message Box.

the short date – returns 3/8/15

the long date – returns Sunday, March 8, 2015

the time -- returns 9:34 AM

the short time – returns 9:34 AM

the long time – returns 7:49:31 PM

set the twelvehourtime to false – sets the time to 24 hr military time in which the time becomes 16:40 rather than 4:40 PM.

the seconds – returns, e.g. 1422542128 (calculated since 1970)

put the ticks – returns, e.g. 85352528513-- a tick is 1/60 of a second (calculated since 1970 as are *the milliseconds*; a millisecond is 1/1000 of a second)

Of course, you most likely would not need to know the number of seconds or ticks since 1970, but you could make use of this information by measuring the differences in time between two events, in a script such as:

on mouseUp

put the ticks into tCounter1

wait one second

put the ticks into tCounter2

put (tCounter2 – tCounter1)

end mouseUp

The Message Box will read 60, confirming that there are 60 ticks in a second. This measurement of differences in time can be used to measure the time it takes to perform any scripting event.

Equivalent scripts:

wait 10 ticks

wait 10 -- by default refers to ticks

Intersect is function useful in certain games. E.g.

if the intersect (btn "target", btn "bullet") is true then answer "Direct hit"

CUSTOM FUNCTIONS

In addition to the functions built into LiveCode, you can create your own. These custom functions are always phrased in the *function()* format, rather than using the word *the*. Why create functions, when you most likely could do the same thing with an ordinary message handler? Example: Say, for simplicity, you want to add 3 numbers (20, 30, and 50), and then put the total into field 1. You could write:

```
on mouseUp
  put 20 into tfirstno
  put 30 into tsecondno
  put 50 into tthirdno
  put (tfirstno + tsecondno + tthirdno) into field 1
end mouseUp
```

Field 1 will show “100”. No problem here. But what if you want to perform this calculation on different sets of 3 numbers throughout the stack on a variety of numbers. You could write the following pair of handlers, with the handler *on myCalc* perhaps residing in the stack script :

```
on mouseUp -- a message handler
  put 20 into tfirstno
  put 30 into tsecondno
  put 50 into tthirdno
  myCalc tfirstno,tsecondno,tthirdno -- Note that 2,3,4 are not placed in
  parentheses here.
  put the result into field 1
end mouseUp
```

```
on myCalc num1,num2,num3 -- a message handler in the stack script
  put num1 + num2 + num3 into myTotal
  return myTotal
end myCalc
```

The numbers *num1*, *num2*, and *num3* are termed **parameters**, each of which is a value, in a sense a kind of variable. Note that the above second handler is an ordinary *on* message handler, and the line *return myTotal* places *myTotal* into a system container called *the result*.

But *the result* is like the variable *it* we discussed previously; it can get lost. Nothing is done with *the result* unless the first handler asks for *it* (*put the result into field 1*). If you forgot to write *put the result* in the first handler, the first handler wouldn't find out about *the result*. And even if the first handler does write *put the result*, it had better do so immediately, since lots of other script lines can call for

the result, which may change, so if the request for *the result* is written too late in the script, there is the possibility that a different *result* might replace the original.

This problem is avoided with function handlers, where the result is **automatically** returned immediately to the original handler:

```
on mouseUp -- a message handler
  put 20 into tfirstno
  put 30 into tsecondno
  put 50 into tthirdno
  put myCalc (tfirstno,tsecondno,tthirdno) into field 1 -- Note that
  -- tfirstno,tsecondno,tthirdno are enclosed in parentheses here, because myCalc
  is now a function.
end mouseUp
```

```
function myCalc num1,num2,num3 -- a function handler in the stack script
  put num1 + num2 + num3 into myTotal
  return myTotal
end myCalc
```

The difference between using the function handler as opposed to a second message handler is that *myTotal* is automatically returned from the function handler to the *mouseUp* message handler without having to rely on the *put the result* line. It is as if the function in the first handler is saying “Do this and get the result back to me immediately so that my script handler doesn’t have to request the result separately”. This may be no big deal for simple scripts, but can avoid confusion in larger, more complex scripts.

Functions don’t have to be calculations, and their parameters don’t have to be numbers; they can be variables that contain letters or words. For instance (again, a simplistic example, for visualization):

Say there is a field “Over the Cliff” and another field “Name” that has two lines that read:

Dover, Eileen
First, Hugo

Consider the following button script:

```
on mouseUp
  put item 1 of line 2 of field "overtheclass" into tname1
  put item 2 of line 2 of field "overtheclass" into tname2
  put arrangeName (tname1,tname2) into field "Over the Cliff"
end mouseUp
```

and its corresponding stack function script, which can be called into play

regardless of what *tname1* and *tname2* are:

```
function arrangeName firstName,secondName  
  put secondName && firstName into fullName  
  return fullName  
end arrangeName
```

Field “Name” will then read “Hugo First”, not of earth-shattering use, but you can imagine the possibility using this scripting method for much more complex and helpful purposes.

(The “&” in the above script is called a concatenation and simply means “plus the following”. In “&&”, the extra “&” means “also add a space”).

.....
A function call doesn’t have to have parameters listed between the parentheses, but it still needs parentheses. For example:

```
on mouseUp  
  put myCombo() into field 3  
end mouseUp  
  
function myCombo  
  put field 1 && field 2 into tHolder  
  return tHolder  
end myCombo
```

Personally, for what I do, which is not very complex, I find it less confusing and less subject to error using ordinary message handlers rather than functions, but different strokes for different folks.

CHAPTER 14. MATH SCRIPT WORDS

add (+)
subtract (-)
divide (/)
multiply (*)

Examples:

add 5 to field “calculation” – If field “calculation” starts out empty, LiveCode assumes it has 0 in it.

```
put 12 into tCounter  
subtract 4 from tCounter
```

divide tCounter by 2
multiply tCounter by 10

It is simpler to just use mathematical symbols (+, -, /, and *), rather than words for many mathematical operations. Thus:

put ((field “calculation”) + 5) into field “calculation”
put (tCounter – 4) into tCounter
put (tCounter/5) into tCounter
*put (tCounter*8) into tCounter*

As in algebra, parentheses are frequently necessary to define the order of the calculation you are seeking to perform. For instance:

5 + 4 * 3 returns 17. LiveCode multiplies 4*3 before adding the 5, as in the standard rules of algebra. Perhaps, though, you really meant to add the 5 and 4 first before multiplying by 3? In this case you need to tell LiveCode to carry out the operation in the order you want. To do this, place parentheses around those parts of the calculation you wish to work out first:
(5+4) * 3 returns 27. It multiplies 9 * 3. This is the standard way of notating calculations.

Parentheses also make for easier reading even at times when they are not necessary. For instance, the following two lines are correct scripting and mean the same thing, but the second line reads more clearly:

put the number of cards in this stack into tCardNumber
put (the number of cards in this stack) into tCardNumber

Parentheses can't hurt.

the number

The number can be used as a property, as in:

the number of this card -- e.g., in a stack of 10 cards, this card might be card number 4.

the number of button “ClickMe” indicates the stacking position of the button in relationship with other buttons on the card. For instance, among the various objects on a card, there might be 10 fields and 5 buttons, with the buttons farther above the card than the fields, and button “ClickMe” might be the 3rd in the stacking order of the buttons, so it is button number 3.

The number can also be a function when referring to a quantity. Examples:

the number of cards in this stack
the number of lines in field "MyText"

the value

The value can be used in reference to text. For instance, in a locked field, the script *put the clickline* may return "line 1 of field 2", while *put the value of the clickline* returns the actual text of the line.

Similarly, in regard to numbers, you might have a script like:

ask "What numbers would you like to multiply?"
put it

If you had typed in **6*9** in the ask dialog box and clicked the "OK" button, the Message Box would just say **6*9**. However, if you scripted it as:

ask "What numbers would you like to multiply?"
put the value of it

then the Message Box would indicate **54**. Here use of *the value* calculated the expression the user intended (6*9) and returned the result 54.

the random

Use *the random* to generate random numbers. Example:

put the random of 10 -- or *put random(10)* randomly generates a number from 1 through 10.

the round

The round rounds off numbers to the nearest whole digit. Examples:

the round of 12.4 -- returns 12.
the round of 12.5 -- returns 13, the next number up.
the round of (-12.5) -- returns (-13), the next number down

the numberFormat

There are times when you want to produce a number with many decimal places to the right for accuracy. At other times, you may just want to list the number in dollars format, with only two decimal places to the right, for cents. In each message handler in which you describe a calculation, you should first indicate the *numberFormat* that you would like to use (unless, of course, you are OK with the default *numberFormat*). Once the message handler ends, LiveCode

automatically reverts to the default *numberFormat*, so you have to declare the *numberFormat* in each script handler that does a calculation if you want to use a special *numberFormat*.

The default *numberFormat* is "0.#####". This means the **maximum** number of decimal places to the right of the decimal is 6.

If the result of a calculation is 1.230456789, the script *set the numberformat to* "<numberformat>" (the numberformat must be in quotes) prior to the calculation shows the calculated number in different ways:

```
numberformat "0.#####": 1.230457
numberformat "0.###": 1.23 (a "0" at the end would be irrelevant)
numberformat "#.00": 1.23
numberformat "##.00": 01.23
```

The number of #s after the decimal in the numberformat indicates the MAXIMUM no of decimal places to the right of the decimal that can appear in the result.

The number of 0s after the decimal in the numberformat indicates PRECISELY the number of decimal places to the right of decimal that should appear.

The number of #s (or 0s) before numberformat decimal indicates the MINIMUM number of digits to the left of decimal that can appear. Hence 1.23 appears as 01.23 for numberformat "##.00"

#.00 or 0.00 is the general format used for money.

The bottom line:

- Use #.00 for the dollar format, which will always be calculated to 2 decimal places.
- In general, keep 0.##### as the default, which will calculate up to 6 decimal places if needed.

average

The *average* is a function that takes the mean of the list of number parameters enclosed in parentheses. Examples:

average (5,12,37) -- returns 18

average (*tList*) -- *tList* being a variable that contains a set of number parameters separated by commas.

abs

The *abs* (absolute) refers to the magnitude of the number regardless of whether it is positive or negative. Thus:

abs (12) -- returns 12
abs (-12) -- also returns 12

= (equals)

<= (is equal to or less than)

> (is greater than)

>= (is equal or greater than)

Examples:

5 = (4 + 1) -- returns "true"
5 = (3 + 1) -- returns "false"
5 > 4 -- returns "true"
5 >= 4 -- returns "true"
5 <= 4 -- returns "false"
if 5 < 6 *then beep* – will get a beep sound

There is

These are not technically math words but are related. The phrases *there is* and *there is no* are very useful when you want to check whether a particular object exists before carrying out a script. For instance,

```
on mouseUp
  go to next card
  if there is a field "data" then put <something> into field "data"
end mouseUp
```

If you had instead just written:

```
on mouseUp
  go to next card
  put <something> into field "data"
end mouseUp
```

then this would result in an error message if no field "data" exists.

first, second, last

Examples:

go to first card – equivalent of *go to card 1*
go second – equivalent to *go to card 2 of this stack*
go last – goes to the last card in the stack

CHAPTER 15. CONSTANTS

Constants are labels that refer to specific and unchanging values. They serve as a shortcut when writing scripts. You can define your own constants or simply use those supplied by Livecode.

True/False

True and *false* evaluate whether a property is turned on or off, or whether a statement is *true* or *false*. Examples:

The locktext of field "data" -- evaluates to either *true* or *false*
 $5 + 4 = 7$ -- evaluates to *false*

The word *true* can often be eliminated in the following sort of statement:

if <something> is true then.....

This is equivalent to:

if <something> then.....

By not specifying whether <something> is *true* or *false*, LiveCode in such examples defaults to *true*.

up/ down /left/ right

The script words *up* and *down* commonly signify whether a keyboard key is up or down, but can also refer to the state of the mouse button. Examples:

wait until the mouse is down
if the optionKey is down then <do something>

Up, *down*, *left* and *right* also have a special use in relation to the *arrowKey* (in cases where the keyboard has arrow keys), as in the following card script:

on arrowKey MyKey
if MyKey is right then go next
if MyKey is left then go prev
if MyKey is up then <do this>
If MyKey is down then <do something else>

end arrowKey

empty

Empty refers to the state of a field or other container. Examples:

put empty into field 1 – deletes all of the field's contents

put empty into tHolder – the tHolder variable then contains nothing

&

&&

& and &&, termed **concatenations**, are used to connect different strings and variables. For instance:

put "Congratulations, " & gName & ". You have passed the exam." into field "Diploma."

In the above example, *gName* might be a variable that contains the name of the individual taking the exam, let's say Bob Smith. Field "Diploma" will display the words, "Congratulations, Bob Smith. You have passed the exam." The "&" is the connector between the strings and the *gName* variable. Note the space put in between the first comma and its following quotation mark, indicating the natural space between the comma and the name. You could also write the script without that space as:

put "Congratulations," && tName & ". You have passed the exam." into field "Diploma"

The extra "&" just adds a space.

Sometimes, when writing a very long line of script, which would awkwardly extend far beyond the right side of the script editor, you might want to break up the line in the script to wrap around to the next line. But you don't want to confuse LiveCode into thinking that there are two separate lines of script. For this you can use the **backward slash** \. E.g.:

*put "Congratulations," && tName &\
". You have passed the exam." into field "Diploma"*

The "\" tells LiveCode that the script line is not finished, and extends onto the next line. If you use the "\", be sure you do not use it between quotation marks. Otherwise Livecode would see the backslash as part of the string and not an operator indicating that the script line is split. Examples:

put "Congratulations, you have just\"

passed the exam" into field "Diploma" -- BAD

*put "Congratulations, you have just passed the exam"\
into field "Diploma" -- GOOD*

Return

Say you want to direct a script to put several lines into a field. Each line would contain a carriage return at its end, forming a new paragraph. You could write:

*put "1. First check the airway." & return & "2. Then check the breathing."\
& return & "3. Then check the circulation." into field 1*

If you wanted an extra blank line between the numbered steps, you can write *return & return* instead of just a single *return*.

The abbreviation *cr* (for carriage return) can be used instead of *return*.

Quote

What if you want the script to place into a field a series of words containing quotes. For instance, you want the field to read:

Bob said, "Let's go home."

You can't just write a script saying:

put "Bob said, "Let's go home."" into field 1

LiveCode would get confused, since there are too many quotation marks. You could approach this in two ways.

1. You could change the quote marks around "Let's go home." to apostrophes, so the script reads:

put "Bob said, 'Let's go home.'" into field 1 -- Good

Or you could keep the quotation marks in the form of the script word *quote*:

put "Bob said, " & quote & "Let's go home." & quote into field 1

CHAPTER 16. IF-THEN-ELSE AND REPEAT STRUCTURES

The *if-then-else* structure is a powerful tool that enables the construction of conditional statements within scripts. Examples:

```
if there is a field "data" then  
  put "12" into field "data"  
  beep  
end if
```

The *end if* part is needed if there are more than one command line. *End if* informs LiveCode that the sequence of conditional directions after *then* has ended.

If there is only a single command, though, one can shorten everything to just one line and there is no need for an *end if*. For example:

```
if there is a field "data" then put "12" into field "data" -- Good
```

For clarity, some programmers feel it is always clearer to use the *end if*, using 3 lines of code, even if the statement could technically be written on one line. Thus,

```
if there is a field "data" then  
  put "12" into field "data"  
end if
```

else

The *else* word is used when an alternative choice is introduced into the script. Example:

```
on mouseUp  
  if there is a field "data" then  
    put "12" into field "data"  
  else  
    go next  
  end if  
end mouseUp
```

Alternatively:

```
on mouseup  
  if there is a field "data" then put "12" into field "data"  
  else go next  
end mouseUp
```

if-then statements can be nested. For instance,

```

on mouseUp
    if there is a field "data" then
        if field "data" is empty then
            put "12" into field "data"
            beep
        end if
    end if
end mouseUp

```

Pearl: Pressing the Tab key while in the Script Editor automatically indents the script lines neatly. If the lines don't align, suspect something wrong with the script syntax.

and/ or

The words *and/or* enable greater versatility in creating conditions for *if-then-else* statements. Example:

```

if tCounter is 10 and (tColor is "red" or tColor is "blue") then beep

```

In the latter statement, a beep will not occur unless *tCounter* is 10 and *tColor* is either red or blue. Do not confuse the word *and* with the concatenation &. The latter is just used to connect strings or add spaces (**Chapter 15**).

```

repeat <number>
repeat with/ repeat for each
repeat until
repeat while
next repeat
exit repeat

```

repeat <number>

The *repeat* structure is used in scripts where you want to repeat an action a number of times. Like *if-then-else* statements with their *end if*, *repeat* must have an ending, *end repeat*, signifying the end of the *repeat* directions. For example:

```

repeat 10 times -- or just repeat 10
    wait 1 second
    go to next card
end repeat

```

The above script takes you to the next 10 cards in succession, each time with a delay of 1 second.

There are several variations on the phrasing of a repeat structure: Examples:

repeat with/ repeat for each

```
repeat with x = 1 to the number of lines in field 1  
  put (line x of field 1) & return after tLineHolder  
end repeat  
put tLineHolder
```

The above script does not specifically state the number of times to carry out the commands, because the scriptor may not know the number of lines in field 1 when writing the script. The script requests that LiveCode determine the number of lines in field 1 and then carry out the directions for each line in succession, putting the information line by line into the variable *tLineHolder*, and then putting *tLineHolder* into the Message Box.

Repeat for each enables a script to run faster through a list than *repeat with*. E.g.

```
repeat for each line x in field 1  
  put x & return after tLineHolder  
end repeat  
put tLineHolder
```

Repeat with and *repeat for each* are even faster if you do not work directly with the lines in the field, but first put the contents of the field into a variable. E.g.

```
put field 1 into tFieldList  
repeat for each line x in tFieldList  
  put x & return after tLineHolder  
end repeat  
put tLineHolder
```

Sometimes you have a stack with many cards and you want the script to go to each card in succession to extract certain information, prepare it as a list, and then do something with the compiled list. While you could use the format:

```
push card  
repeat with x = 3 to the number of cards  
  go to card x  
  put field 1 of card x & return after tInfoHolder  
end repeat  
pop card  
<then use tInfoHolder in some way on the originally pushed card>
```

This script can unfold much faster by not traveling directly to the cards:

```
repeat with x = 3 to the number of cards
  put field 1 of card x & return after tInfoHolder
end repeat
<then use tInfoHolder in some way on the original card>
```

If you do want to actually go from one card to the next through the stack, this will go much quicker by first setting lockscreen to true (*set lockscreen to true* or *lock screen*). In that way all the action is done behind the scenes without having to add time by visually showing each card in succession.

Sometimes, when there is a repeat loop that takes a long time to act, you may want to add some sort of progress indicator to let the user know that, yes, the script is progressing and the program did not freeze. This could be an animated button, a busy cursor, a text field that progressively indicates the number of times the loop has repeated, or a progress bar.

Unfortunately, for a long process, the overhead of showing this progress also adds time to the process. Depending on what you are trying to do you may want to take this into account when choosing how to display this progress. In general, the speed of an animated button is quicker than a busy cursor, which in turn is quicker than a text-based progress field, which in turn is quicker than a progress bar. (Thanks to Sarah Reichelt for these speed suggestions.)

A variation in the form of a countdown to blastoff:

```
on mouseUp
  repeat with x = 10 down to 1
    put x into msg
    wait 1 second
  end repeat
  put "Blast off!!"
end mouseUp
```

repeat until

An example:

```
on mouseUp
  put 0 into tCounter
  repeat until tCounter = 30
    put (tCounter + 1) into tCounter
    put tCounter into msg -- (or just write put tCounter)
    wait 10 ticks -- or just write wait 10
  end repeat
end mouseUp
```

The first line of the script starts off *tCounter* at 0, *tCounter* being a made-up variable; you could have made up any other word, e.g. *tHummingbird*. (Actually, you don't even have to write the line *put 0 into tCounter*, since LiveCode will by default assume *tCounter* is 0). The script tells the counting process to put the evolving sum after every 10 ticks into the Message Box, until *tCounter* reaches 30.

Repeat until can also be used in other contexts:

```
on mouseUp
    repeat until the mouse is down
        <perform some process>
    end repeat
end mouseUp
```

The value of a handler such as the above is that it provides the user a way of interrupting a script by pressing the mouse down.

repeat while

An example:

```
on mouseDown
    put 0 into tHolder
    repeat while the mouse is down
        put (tHolder + 1) into tHolder
        put tHolder
    end repeat
end mouseDown
```

This script will put a continuous counting sequence into the Message Box, continuing as long as the mouse is down.

next repeat

Sometimes when operating on a collection of things within a repeat loop you want to leave out some of the repeats for some reason. This is where you would use *next repeat*.

For example, say you write a repeat structure directing the script to go to each card in the stack and issue a beep on each card except if the card is titled "quiet". You might invoke *next repeat* in the script as follows:

```
on mouseUp
    repeat with x = 1 to the number of cards
        go to card x
```

```

        if the short name of card x is "quiet" then next repeat
        beep
        wait 10
    end repeat
    answer "I'm through beeping."
end mouseUp

```

The above script will *beep* on every card it goes to, except for the card named "quiet", since the script directs the *repeat* to bypass that card and go on to the next repeat in the loop. Once the *repeat* process ends (on the last card in the stack), the script directs LiveCode to go on with the next step, namely announce that it is through beeping.

exit repeat

There can also be times when you want to leave the repeat loop early. For this you would use *exit repeat*. For example, if you want to go to each card in succession, but stop when you come to that card named "quiet". You might write:

```

on mouseUp
    repeat with x = 1 to the number of cards
        go to card x
        if the short name of card x is "quiet" then exit repeat
        wait 10
        beep
    end repeat
    answer "I'm through beeping."
end mouseUp

```

In the latter script, there will be no more beeps once the card "quiet" is reached. The script directs LiveCode to stop the looping at the "quiet" card and announce that it is through. *Exit repeat* doesn't just direct the script to loop back to the next *repeat*. It stops the entire *repeat* process.

Repeat statements can be nested, just as can the *if-then-else* statements. Example:

```

on mouseUp
    repeat with x = 1 to the number of cards in this stack
        go to card x of this stack
        repeat with y = 1 to the number of flds in card x of this stack
            put empty into fld y of card x of this stack
        end repeat
    end repeat
end mouseUp

```


The above script could be abbreviated, though:

```
on mouseUp
  repeat with x = 1 to the number of cds
    go cd x
    repeat with y = 1 to the number of flds
      put empty into fld y
    end repeat
  end repeat
end mouseUp
```

Eliminated in the script are the words *in this stack*, *of this stack*, *in card x of this stack*, and *of card x of this stack*. You don't need these words, since LiveCode by default assumes the cards you are talking about are those belonging to this stack, and the fields you are talking about refer to the card you are on (card x).

Sometimes you can get into an endless loop by using *repeat* without specifying for how long. Pressing command/period (control/period in Windows) will terminate any script, provided the stack **Property Inspector/Basic Properties** has “**User can't abort scripts**” unchecked (*cantAbort* is set to *false*).

is/ is not/ contains/ is among

Imagine a card with a single field, containing the text:

“I do believe that I am a field.”

Now consider these scripts, delivered perhaps through the Message Box.

```
if word 2 of field 1 is “do” then beep -- You get a beep. Other scripts:
if word 1 of field 1 is not “do” then beep - get a beep.
if field 1 contains “believe” then beep -- get a beep
if field 1 contains “bel” then beep -- get a beep
if field 1 contains “ieve” then beep -- get a beep
if field 1 contains “believe that” then beep -- get a beep
if field 1 contains “believe am” then beep – no beep
```

The last line does not produce a beep, even though the words “believe” and “am” are in the field, because “believe” and “am” are not together as a single string.

```
if “believe” is among the words of field 1 then beep -- get a beep
if “believe that” is among the chars of field 1 then beep -- get a beep
if “believe that” is among the words of field 1 then beep -- no beep, because
LiveCode in this script line is looking here for a single word, not a combination
of words or characters.
```

pass

The word *pass* in a script enables the script's message handler to pass through to the next level of the hierarchy. For instance, say button 1 has the script:

```
on mouseUp  
  beep  
  pass mouseUp  
end mouseUp
```

and that the card on which button 1 resides has the script:

```
on mouseUp  
  answer "Quiet please!"  
end mouseUp
```

Without the line *pass mouseUp*, the button would trap the *mouseUp* and the *mouseUp* message would not get to the card. All that would happen would be a beep. However, by including the *pass mouseUp*, the message also passes through to the card, which results in the "Quiet, please!" answer dialog, in addition to the beep.

We also discussed the word *pass* in our example with trapping keystrokes where we had a field for text input with the following script:

```
on keyDown MyKey  
  if MyKey is not a number then answer "You must enter a number"  
  else pass keyDown  
end keyDown
```

Without "*pass keyDown*" the actual keystroke would be trapped in this handler and the keystroke character would never make it through to the field.

CHAPTER 17. CURSOR SCRIPTING

Cursor

The default appearance of the cursor is the crossed arrow when in Edit Mode, and the uncrossed arrow when in Run mode. The cursor in Run mode, by default, changes to an I-beam when over an unlocked field (a field in which it is possible to type).

One can set the cursor appearance to a number of shapes that are embedded in LiveCode:

arrow -- default
none -- no visible cursor
busy -- spinning beachball, indicating a process under way
watch -- also indicates a process under way
cross -- used during painting, drawing or selecting a small area
hand -- a finger pointing up
iBeam -- the default for selecting text in an unlocked field

The cursor appearance is changed in a script by “setting” it. Example:

lock cursor -- or *set lockCursor to true*
set cursor to busy

The *lock cursor* command is important before setting the cursor; otherwise the cursor will not keep its new shape, but immediately revert back to its default arrow shape. Once a cursor is locked, though, the cursor maintains its new appearance, even after the script is finished. You can have the cursor revert to its normal default through the command *unlock cursor* (or *set lockCursor to false*).

The latest versions of LiveCode enable you to use any image as a cursor, by referring to it by its ID number. E.g.,

lock cursor
set cursor to 493

Cursor images can also be imported into the Image Library (via the LiveCode menu bar under **Development/ Image Library**), where they will remain for future projects.

Sometimes, for inexplicable reasons, you find that the cursor has changed to something not the usual arrow shape. To correct this, just type *set cursor to arrow* in the Message Box.

hotSpot

The *hotSpot* of a cursor is the exact point on the cursor that you want to act as the point that clicks on the target. By default, that point is 1,1, but you can set it to other points through a script. Examples:

set the hotSpot of image ID 1008 to 5,5
set the hotSpot of image “arrow” to 6,10

CHAPTER 18. PRINTING

When working with a stack, you can use the LiveCode menu, under **File/ Print Card** or **File/ Print Field** to do a simple printing of a card or field. You can specify in the printing dialog how many copies you wish to print.

If a card has a scrolling field, and you need to scroll to see all the text, **Print Card** will only reveal the text that is visible at the time of printing. Hence, the value of *revPrintText* in a standalone; it prints the entire contents of the field.

RevPrintText

To use a script to print all the contents of a scrolling field titled “MyData”:

```
revPrintText field “MyData”
```

Also, *revPrintText* prints other things besides fields:

```
revPrintText “Hello there”  
revPrintText tmyVariable
```

print

If you use scripting to ask for a printing of a card or stack, use these scripts in the following circumstances:

To print the entire stack:

```
print all cards  
or  
print this stack
```

To print a single card:

```
print this card  
print card 3
```

print marked cards -- prints those card that have been marked (i.e., the **marked** box in the card Property Inspector is checked).

answer page setup / open printing with dialog

Answer page setup brings up the **FILE/Page Setup** box, enabling you to set the orientation and scale of the printing.

Open printing with dialog (on Macintosh; on Windows, use *answer printer*) enables you to select how many copies of a stack or card to print and sets up how many cards to print per page:

```
on mouseUp
```

```
answer page setup
open printing with dialog
print this stack -- or print this card
close printing
end mouseUp
```

The *close printing* part is necessary on Mac because it tells LiveCode to actually go ahead with the printing. On Windows, use *answer printer* without *open printing with dialog* or *close printing*.

printMargins

The *printMargins* sets the width of the margins of the page when printing. The numbers are given in pixels, assuming 72 dots per inch (dpi).

set the *printMargins* to 18,18,18,36 -- the four numbers refer to the left, top, right, and bottom margins of the page respectively.

printPaperScale

PrintPaperScale sets the magnification of the printing. Numbers between 0 and 1 represent 1 to 100% magnification:

set the *printPaperScale* to .8 – 80% magnification
set the *printPaperScale* to 2 – 200% magnification

CHAPTER 19. INTERNET COMMUNICATION

LiveCode is extremely powerful in connecting with the Internet. The *launch URL* command provides a direct way to go to a specific web page, by simply typing the web page's URL after *launch URL*. Example:

launch URL "<http://www.google.com>" -- opens Google in the default browser
To learn about Florida birds, you could type **Florida Birds** in the Google search box. Google would then list a series of articles, using the browser search path:

https://www.google.com/?gws_rd=ssl#q=florida+birds

You could do all this directly from LiveCode by using the script:

```
on mouseUp
launch URL https://www.google.com/?gws\_rd=ssl#q=florida+birds
end mouseUp
```

You can carry this a step further using Google Images: If you type **Florida Birds** into the Google Images search engine box, a whole collection of pictures of Florida birds appear. But look at the browser's search path; it reads something like:

http://www.google.com/search?num=10&hl=en&site=imghp&tbm=isch&source=hp&biw=1163&bih=1150&q=Florida+birds&oq=Florida+birds&gs_l=img.3..0l2j0i24l8.9421.11690.0.12577.13.12.0.1.1.0.84.939.12.12.0...0.0...1ac.jWKishz9l3c

That's a lot of gobbledey gook mixed with the actual search words "Florida birds" with a plus sign between "Florida" and "birds". LiveCode can be programmed to automatically do the entire search from a single click from within LiveCode (the following launch URL command is all on one line):

on mouseUp

launch URL

http://www.google.com/search?num=10&hl=en&site=imghp&tbm=isch&source=hp&biw=1163&bih=1150&q=Florida+birds&oq=Florida+birds&gs_l=img.3..0l2j0i24l8.9421.11690.0.12577.13.12.0.1.1.0.84.939.12.12.0...0.0...1ac.jWKishz9l3c

end mouseUp

By substituting other words for "Florida+birds", you can create a script within LiveCode that enables the user to select any word(s) in a list field and do an immediate Internet search for all the literature or all the images. This has significant potential value to educators who wish to introduce Internet searches into their courses, using LiveCode.

This same technique was used to prepare a standalone listing the known infectious diseases, over 10,000 of them, including images and literature. See Atlas of Human Diseases, available as a free standalone download from <http://medmaster.net/freedownloads.html>. This task would take a lifetime in the old days for an ordinary print book, but took only a few hours of programming using LiveCode, once the list of diseases was prepared.

A LiveCode stack or built (standalone) application can be uploaded to one's website, where a user can download it to the user's computer. Downloading a standalone, rather than a stack, has an advantage in that the user who does not have LiveCode installed does not require the LiveCode player or Internet access to view the standalone after it is downloaded.

A stack prepared on Macintosh will run on Windows, and vice versa, provided you have LiveCode installed.

CHAPTER 20. SPECIAL EFFECTS SCRIPTING

Apart from excellent pictures and interesting looking buttons, you can dress up the appearance of a stack with special effects that take place on opening a card:

Transitional effects

visual effect “barn door close” or “barn door open”

visual effect “checkerboard”

visual effect “dissolve”

visual effect “iris close” or “iris open”

visual effect “plain”

visual effect “push up”, “push down”, “push right”, or “push left”

visual effect “reveal up”, “reveal down”, “reveal right”, or “reveal left”

visual effect “scroll up”, “scroll down”, “scroll right”, or “scroll left”

visual effect “shrink to bottom”, “shrink to center”, or “shrink to top”

visual effect “stretch from bottom”, “stretch from center”, or “stretch from top”

visual effect “venetian blinds”

visual effect “wipe up”, “wipe down”, “wipe right”, or “wipe left”

visual effect “zoom close”, “zoom in”, “zoom open”, or “zoom out”

The visual effect should be declared before the command to go to the card.

Examples:

on mouseUp

visual effect “barn door close”

go next

end mouseUp

The speed of the transitional effect can be specified as *normal*, *slow*, *fast*, or *very fast*. Example:

visual effect “checkerboard” slow

You can also add an audioclip to the visual effect:

visual effect “checkerboard” slow with sound “Whoosh.aif”

You can expand upon the number of visual effects in your repertoire by calling up QuickTime’s special effect dialog box:

answer effect – brings up the Quicktime dialog box, from which you choose the effect you want, which is put into the variable *it*.

set the myStackEffect of this stack to it -- saves the QuickTime visual effect as a custom property of the stack for future use. Then, the effect can then be called from a handler, e.g.:

```
on mouseUp
  visual effect the myStackEffect of this stack
  go next
end mouseUp
```

Transitional effects can also be applied to objects on a card, when showing or hiding an object. For instance, if a hidden image is titled “Sunflower”, it can be made gradually visible with a special effect:

```
show image “Sunflower” with visual effect “dissolve”
show image “Sunflower” with visual effect the myStackEffect of this stack
```

Human Speech

You can introduce the spoken sounds through the command:

```
revSpeak “How are you today.”
revSpeak field “Lesson 1”
```

If you want a particular male, female, or other voice to speak, and not the default voice, type in the Message Box (on Mac, as different voices do not appear to work on Windows):

```
revSpeechVoices()
```

That will give you a list within the Message Box of the different voices that can be used for *revSpeak*. Once you have picked a voice you like (e.g., “Bruce”), use it in the following script:

```
revSetSpeechVoice “Bruce”
revSpeak “How are you today?” -- it will be Bruce’s voice
```

Interestingly, case sensitive letters, generally not important in LiveCode scripting, are important in typing the voice name:

```
revSetSpeechVoice “bruce” -- won’t work since “B” is not capitalized
```

```
revSetSpeechPitch -- sets the pitch of the speech from 30 to 127 (on Mac)
revSetSpeechSpeed -- sets the speed of the speech from 1 to 300
```

```
on mouseUp
  revSetSpeechVoice “Agnes”
  revSetSpeechPitch 40
  revSetSpeechSpeed 100
  revSpeak “Hello, everybody.”
end mouseUp
```


windowShape

You don't have to be satisfied with the plain old rectangular stack window. A stack can be any shape. If you import into LiveCode an odd-shaped PNG image with its accompanying transparent areas, note the image's ID number in the image's Property Inspector in the **Basic Properties** section. For instance, if the ID number is 1008, you can set the shape of the stack to the shape of the image (the image does not have to be visible) by typing:

set the windowShape of this stack to 1008

To return to the default stack shape, type:

set the windowShape of this stack to none

The only problem with setting the stack to a unique windowShape is that there is no title bar, and the stack cannot be moved manually. You can resolve this problem in several ways: You can type:

set the loc of this stack to the screenloc

This will at least position your stack at the center of the screen. Alternatively, you can write a special stack script (thanks to LiveCode guru and artist Scott Rossi) to allow you to drag the unusually shaped stack manually:

local sgDragging, sgLeftOffset, sgTopOffset

on mouseDown

put item 1 of the mouseLoc into sgLeftOffset

put item 2 of the mouseLoc into sgTopOffset

put true into sgDragging

end mouseDown

on mouseMove

lock screen

if sgDragging is true then

set the left of this stack to item 1 of globalloc(the mouseLoc) - sgLeftOffset

set the top of this stack to item 2 of globalloc(the mouseLoc) - sgTopOffset

end if

unlock screen

end mouseMove

on mouseRelease

put false into sgDragging

end mouseRelease

```
on mouseUp  
  put false into sgDragging  
end mouseUp
```

Many other ideas for unique and beautiful interfaces for LiveCode can be found on Scott Rossi's website at <http://www.tactilemedia.com>.

CHAPTER 21. SCRIPT DEBUGGING

LiveCode in many cases picks up scripting errors and points the programmer immediately to the line of code with the problem. Sometime this an error in the scripting syntax. At other times, the syntax may be correct, but the script is not carried out, e.g. the script makes reference to a nonexisting field or card.

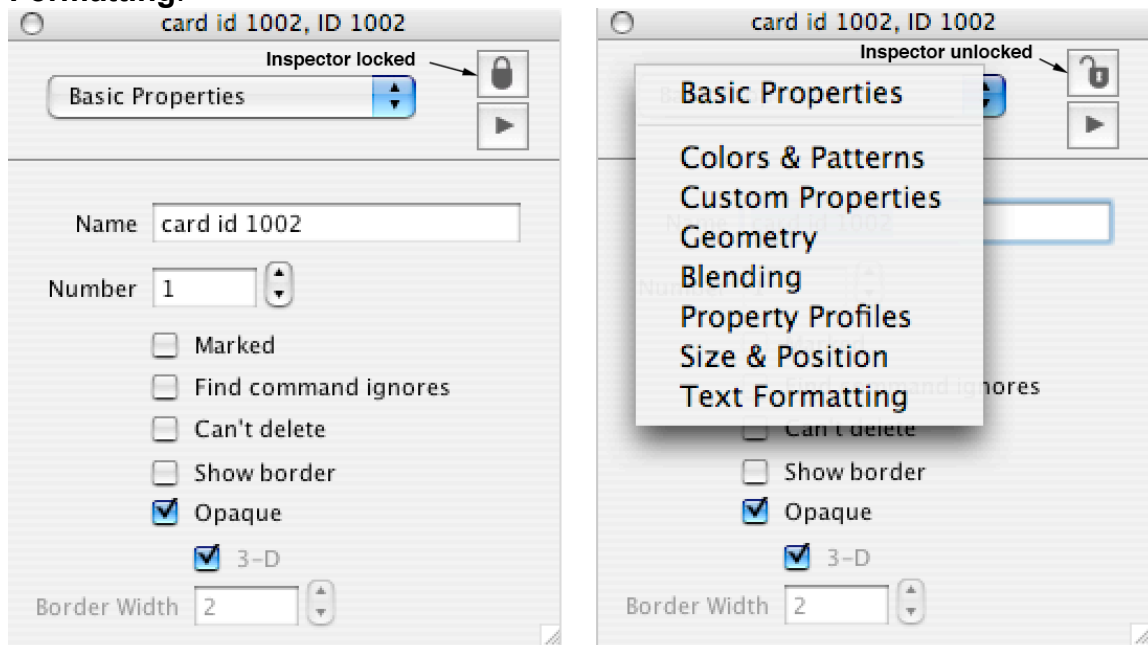
When a script does not work and LiveCode doesn't satisfactorily tell you why, there are several ways of approaching the problem:

1. Check the script for spelling errors, commonly a misspelled variable.
2. Check the script for syntax errors, such as forgetting to put in an *end if* or *end repeat* or beginning or ending quote mark or parenthesis.
3. Be sure you are not using a reserved word.
3. A quick and dirty way to determine whether a script has performed adequately up to a certain point is to place a temporary test command at a point in the script, such as *beep*, *put*, or *answer* to see if the test command executes. If it does, then the script is working up to that point. Be sure to include your own special comment sign after the command (e.g. -- ###) to remind you where the test command is so you can later remove it.
4. Be sure you're testing in Run mode rather than Edit mode.
5. It may help to retype the script line in question; it may contain a hidden character, particularly if it was copied from a word-processing document.
6. Consider an alternative work-around script.

I have found the above approach sufficient for my own projects, but for much longer and complex scripts, LiveCode provides a more professional way of stepping through and examining the script line by line. This will not be discussed here, but a description of LiveCode's debugging process may be found in the LiveCode User Manual.

SECTION 3. PROPERTY INSPECTORS

Property Inspectors enable the modification of many properties of the stack, card, and controls on the card without scripting. Open LiveCode, create a new Main Stack, and take a look at the pulldown menu of the card Property Inspector. You see a number of categories in which you can change the properties of the card (see below): **Basic Properties**, **Colors and Patterns**, **Custom Properties**, **Geometry**, **Blending**, **Property Profiles**, **Size and Position**, and **Text Formatting**.



Card Basic Properties

Every object in LiveCode, including groups, cards, controls, and the stack itself, has its own Property Inspector. The Property Inspectors for the stack, the cards, and controls on the cards have many similarities, but also a number of pertinent differences.

Since many of the properties of the different Property Inspectors are similar, the question naturally arises as to which Property Inspector gains precedence when there is a conflict between them. For instance, the Text Formatting menu of the stack Property Inspector sets the text formatting (e.g. text font and font size) for the stack as a whole. What if the card Text Formatting differs from that of the stack? Which Property Inspector wins out? Answer: The card's text formatting wins out over that of the stack.

Similarly, a field's text formatting has precedence over that of the card. Is there anything that has precedence over the field's text formatting? Yes! Say you have set a field's text formatting (font, font size, and style) through the field's Property Inspector, but you want to change certain words within the field to a different font, font size, style or color than the other words within the field. Then the LiveCode menubar under **TEXT** has precedence over the field's overall text formatting in its Property Inspector, so you can format individual words differently within a field.

The upper right hand corner of each Property Inspector contains a "lock" icon (**See above**). Normally the Property Inspector is unlocked, which means that every time you open a new Property Inspector, the previous Inspector closes, so only one Property Inspector remains open at a time. By locking the Inspector, it remains open, so that more than one inspector can remain open for comparison.

To avoid repetition when discussing the Property Inspectors of stacks, cards, and other objects, since the various Property Inspectors are largely similar, I will confine the discussion to the pertinent differences. For further information, consult the LiveCode dictionary.

CHAPTER 22. THE STACK PROPERTY INSPECTOR

Open the stack Property Inspector by right-clicking on the stack and selecting **Stack Property Inspector**. (If you only have a single button mouse, click with the Control key down.) Look at the pulldown menu at the top of the stack Property Inspector, which enables you to access a number of properties of the stack that you can modify. For the purposes of this introductory book, I will only discuss the most useful features (in my experience).

STACK BASIC PROPERTIES (Fig. 22-1)

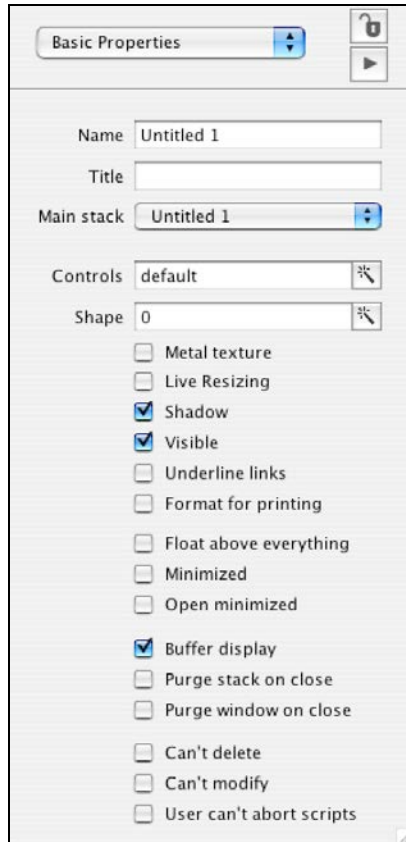


Fig. 22-1. Stack basic properties.



Fig. 22-2. Decorations.

NAME (*name*): The **Name** field is where you give the stack a name, commonly a single word for easy scripting reference. When you refer to a stack in a script you will use that name when referring to the stack. E.g.:

set the name of this stack to "MyStack"
put the name of this stack into message

Do not prefix the name of a stack with "rev". This is reserved for the LiveCode engine. Also, even if you use a single word for the name, always surround names, or other strings of text, with quote marks to avoid conflict with other words, such as variables (discussed later), which never have quote marks.

Note that the script word `set` is used when setting any of the properties of an object via a script.

TITLE (*title*): The **Title** you give the stack is not used in scripting. It is simply the words that you want the user to see in the stack's title bar at the top of the stack. The **Title** may be more colorful than the stack's name, which the user does not see. If you do not give the stack a title, then the stack's name becomes the title by default.

You may have wondered why your stack has an asterisk next to its name in the stack title bar. The asterisk disappears when you have assigned a title to the stack. The absence of the asterisk serves as a reminder to the programmer to use the stack's name, rather than its title, when scripting.

MAIN STACK (*mainStack*): If you have a mainstack "A" and a mainstack "B" open, and no substacks, and you want mainstack "B" to become the substack of mainstack "A", you can change mainstack "B" to a substack of "A" by selecting "A", from "B"'s **Main Stack** pulldown menu. Then, when you open Stack A's Application Browser, you will see that A is listed as the Main Stack, and B as the substack of A. At that point you can discard stack "B.livecode" since you have just duplicated it to be a part of Stack "A" as a substack.

In doing this, bear in mind that a main stack can have a substack, but a substack cannot have its own substack.

CONTROLS (*decorations*): Enables you to select which combination of title bar controls you would like to see, for open, close, or magnify (**Fig. 22-2**).

SHAPE (*windowShape*): If you import an image into LiveCode, particularly a PNG image, which can have transparent areas, note the ID number of the image in that image's Property Inspector. If you then set the Shape of the stack to the image's ID number, the stack will take on the shape of the image. A stack doesn't have to be rectangular. It can have any shape and even have holes in it where the transparencies are located!

VISIBLE (*visible*): Checking the visible box enables you to see the stack. Unchecked, the stack is hidden (not closed, but invisible).

Note that when there is a checkbox involved in a Property Inspector, checking off the box sets the property to *true*, while unchecking it sets the property to *false*. E.g.:

set the visible of this stack to false -- unchecks the **Visible** box and hides the stack

USER CAN'T ABORT SCRIPTS (*cantAbort*): Normally, pressing command/period (Mac) or control/period (Windows) will stop an endlessly looping script. Checking this box (set **cantAbort** to true) will not allow you to stop the script, but will allow you to go nuts trying to decide how to turn off the script. Better leave this box unchecked.

Note that positioning the mouse cursor over a word in the Property Inspector in most cases gives you the script word for setting the property.

STACK COLORS AND PATTERNS (Fig. 22-3)

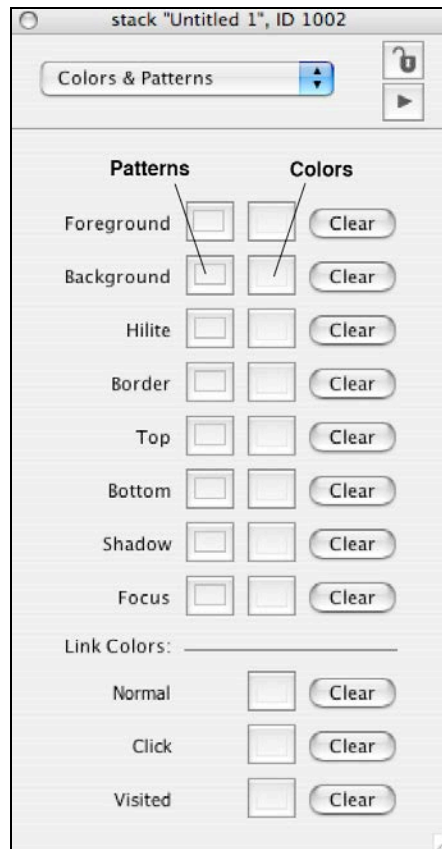


Fig. 22-3. Stack Colors and Patterns.

FOREGROUND (*foregroundColor*): This sets the color (or the pattern) of text in the stack's fields and buttons.

BACKGROUND (*backgroundColor*, *backgroundPattern*): This sets the color (or the pattern) of all the cards in the stack. If you want to select from a wide variety of colors, simply type *the colornames* in the message box. This will provide you with over 500 additional colors to select from.

set the backgroundColor of this stack to Burlywood3

Note that each of LiveCode's color pickers has a small "color grabbing" magnifying glass icon in its upper left corner (**Fig. 22-4**). This can help you select special colors from within or outside(!) LiveCode by clicking on them with the magnifying glass.

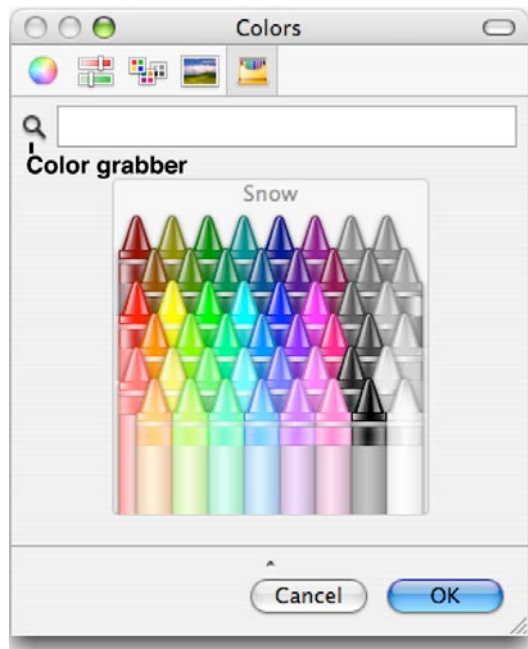


Fig. 22-4. Color grabber.

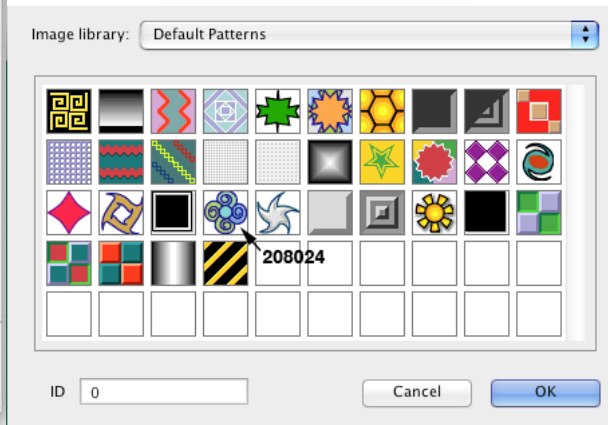


Fig. 22-5. Pattern IDs.

Regarding patterns, you can learn the ID number of the pattern you want by holding the mouse for a few seconds over the background pattern (after clicking on the `backgroundPattern` box in the Colors and Patterns section) (**Fig. 22-5**). This can then be incorporated into a script. E.g.,

set the backgroundPattern of this stack to 208104

HILITE (*hiliteColor*, *hilitePattern*): This is the hilite color (like a hilighting pen) that surrounds text within the stack when the text is selected in a field or a menu.

STACK BLENDING (Fig. 22-6)

BLEND LEVEL (*blendLevel*): Note how sliding the “Blend Level” Bar alters the transparency of the stack. You can make the stack partially or fully transparent if you wish.

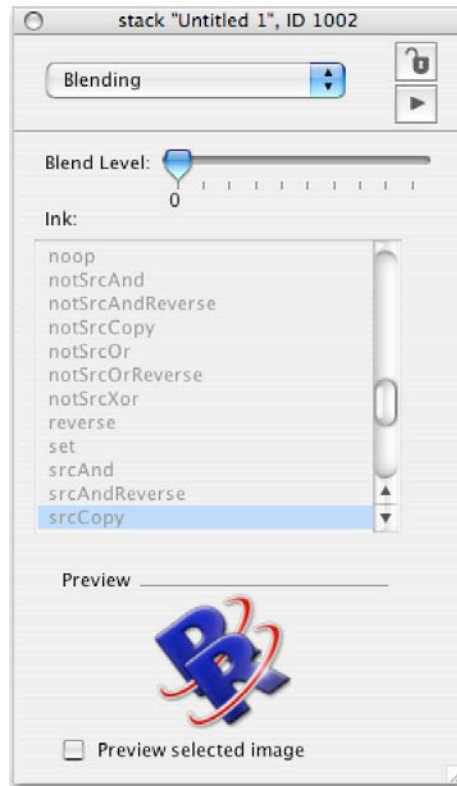


Fig. 22-6. Stack Blending.

STACK SIZE AND POSITION (Fig. 22-6)

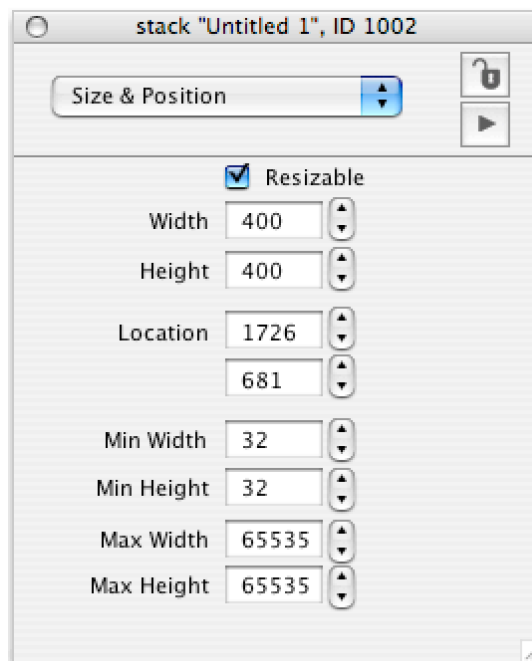


Fig. 22-7. Stack Size & Position.

RESIZABLE (*resizable*): Checking this box enables the user to resize the stack.

If it is unchecked, the little resizable handle at the bottom right of the stack (**Fig. 2-2**) disappears, and the stack cannot be resized by the user.

WIDTH (*width*): Sets the width of the stack.

HEIGHT (*height*): Sets the height of the stack.

LOCATION (*loc*) : This tells you the location of the center of the stack with reference to the x and y coordinates of the monitor. The coordinates of the upper left corner of the monitor are “0,0”, the first number being the x coordinate, and the second number being the y coordinate.

The two fields for the **Location** property of the stack are the x coordinate (the top one) and the y coordinate below it. Together they make up the *loc* of the stack (e.g. a *loc* of “1726,681”). Try moving the stack around by its title bar and you will see how the stack’s *loc* numbers change continuously. A useful scripting word to know here is *screenLoc*. That is the location of the center of the monitor. Thus, to set a stack to the center of the monitor in a script, write in the Message Box:

Set the loc of this stack to the screenLoc

This script, by the way, can be useful in cases where the stack somehow gets stuck way at the top of the monitor and you can’t grab its title bar to reposition the stack manually.

STACK TEXT FORMATTING (Fig. 22-7)

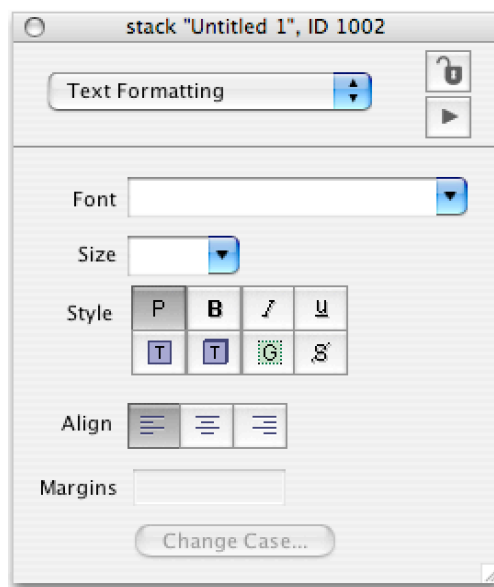


Fig. 22-8. Stack text formatting.

You can set the font and font size for all the buttons and fields in the stack. You

can also set the Style of the text (Plain, Bold, Italic, Underlined) and the text alignment (right, center, or left justified). Note that cards and fields also have the option of text formatting. When there is a conflict, the card setting overrides the stack setting, and the field setting overrides the card and stack. Different fields can have different text properties.

FONT (*textFont*): Sets the type of font for the stack as a whole.

SIZE (*textSize*): Sets the size of the text.

STYLE: Sets whether the font is **Plain**, **Bold**, **Italic**, or **Underline**, with additional options of **Box text**, **3D box text**, **Link text**, and **Strikeout**.

Stack Scripting

open/close vs show/hide

Open stack and *close stack* differ from *show stack* and *hide stack*, even though both sets of commands change the visibility of the stack. When you *hide* a stack, it is still *open* but is just invisible. When you *close* a stack, it actually closes and cannot be seen by just issuing a *show* command. E.g.:

```
close stack "MyStack"  
open stack "MyStack"  
hide stack "MyStack"  
show stack "MyStack"
```

Be cautious how you use these commands in removing a stack or substack from view. If you use *close stack* or *hide stack*, plan ahead with an appropriate *open stack* or *show stack* message respectively to be able to view the stack later.

Show and *hide* are also applied to any kind of control on a card:

```
hide field "data"  
show image "flowers"  
hide btn id 1015  
show player "MyMovie"
```

"Stack" is also very useful as a suffix in the following message handlers:

```
on openStack  
on closeStack  
on preOpenStack
```

For example, in the script of a stack you could write:

```
on closeStack
  <do something awesome>
end closeStack
```

This tells LiveCode to carry out some command(s) at the time the stack is closed. To tell LiveCode what to do when a stack is opened:

```
on openStack
  <do something mind-boggling> -- when the stack is opened and becomes
  visible
end openstack
```

When *openStack* is used, the directions (*<do something mind-boggling>*) are not carried out until the stack is open.

PreOpenStack does the same thing as *openStack*, except that it does it earlier, before the stack is open. This provides you the opportunity to do things behind the scenes (e.g. adjusting the stack's position, or adjusting the appearance of controls on the first card) before the stack is open, like straightening out your house before the company comes.

```
on preOpenStack
  <do something secretive before opening the stack>
end preOpenStack
```

The *preOpenStack* and *OpenStack* commands are automatically sent to every stack that opens. If the stacks have no handlers for these words, nothing happens. However, if the mainstack contains a *preOpenStack* or *OpenStack* handler, then whenever a substack opens it will enact these handlers because of the message hierarchy flow from substack to stack. To prevent this from happening, it is a good idea to put the *preOpenStack* and *OpenStack* handlers in the script of the first card in the mainstack rather than in the stack script itself.

lockscreen

lock screen/unlock screen

go invisible

Suppose you want LiveCode to carry out a script in which the program goes to all the cards in a stack one by one and collects information from each. For instance, there may be a name field on each card and you want to compile a list of all names in the stack. You could, of course, simply direct LiveCode to go to each card in succession, a relatively slow process in which you see each card being flipped as the information is collected. You can speed up the process significantly by setting *lockscreen* to *true*:

```
On mouseUp
  set lockscreen to true -- or, alternatively, lock screen
  <go to all the cards and do something with each behind the scenes and then
  return>
end mouseUp
```

In that way, you suspend all visibility of what is going on while the cards are visited. The screen never refreshes until the end of the process, and you don't see the flipping of one card after another. The system automatically becomes unlocked again after the script is finished. Processes involving traveling to many cards are carried out faster when *lockscreen* is set to *true*. If you want to unlock the screen sometime in the middle of a script, then at that point use:
set lockscreen to false -- or, alternatively, *unlock screen*.

start using

Part of the message hierarchy flow is from substack to mainstack. It normally does not go from one substack to another, or to a mainstack other than its own. If you want to have the message flow go to another substack or mainstack, the *start using* command can be useful. E.g., if the second stack (say, **MyOtherMainstack**) contains within its stack script:

```
On MySpecialCommand
  <do something special>
end MySpecial Command
```

then you could write in your first stack:

```
On mouseUp
  start using stack "<MyOtherMainstack>"
  MySpecialCommand
end mouseUp
```

This can be useful in accessing the scripts of another stack.

Of course if you are using the scripts in another stack repeatedly, it may be a good idea to have the "start using" command appear in an *on preopenStack* handler of your mainstack, so that you can use it throughout the application you are developing.

backdrop

Setting the *backdrop* enables you to introduce a background color or pattern to the entire screen behind the stack, removing from view other distracting elements of the desktop. For example, to place a backdrop of a particular color:

set the backdrop to black
set backdrop to none -- removes the backdrop
set the backdrop to 153,255,51

It is also possible to set the backdrop with a pattern (patterns can be found in the **Colors and Patterns** section of any Property Inspector). To find the ID of a pattern, simply hold the cursor over the pattern in the Property Inspector until the pattern's ID number appears as a tooltip). LiveCode has over 150 of them. You might want to try some. Example:

set the backdrop to 208007

palette
topLevel
modeless
modal

Chapter 2 discussed the 4 general types of stack windows: **topLevel**, **modal**, **modeless**, and **palette**. One stack type can be converted to another through scripting.

palette stack "MyPersonalToolsStack"
go to stack "MyPersonalToolsStack" as palette -- an alternative script
open stack "MyPersonalToolsStack" as palette -- still another alternative
topLevel stack "MyRegularStack"
modeless stack "MyPreferencesStack"
modal stack "MyCantLeaveSoEasilyStack"

Be careful in creating a modal stack! It has no close box and you can't leave the stack until the user provides a response that allows the stack to close. So you might want to include a button on the stack with a script such as *close this stack* to enable the user to leave.

If, during development, you want to get rid of an unwanted substack, you can do this through the Application Browser (**TOOLS/APPLICATION BROWSER**). Right click on the stack in the Application Browser and select *Delete Substack* from the menu that appears.

go to stack <> in stack <>

Sometimes, when you are in stack "A" and then open .livcstack "B", you don't want to continue to see stack "A". Rather, you'd like stack "A" to close and stack "B" to take its exact place. This can be done with the script:

go to stack "B" in stack "A"

If you want to reopen the original stack “A”, without seeing stack “B”, simply type:

go stack “A” in stack “B”

“Going to” a stack opens that stack.

CHAPTER 23. THE CARD PROPERTY INSPECTOR

You’ll know it’s a card Property Inspector, because it says “Card” on the Property Inspector’s title bar, which also lists the card’s ID number. Do not confuse this with the Stack Property Inspector, which sometimes pops up unexpectedly.

CARD BASIC PROPERTIES (Fig. 23-1)

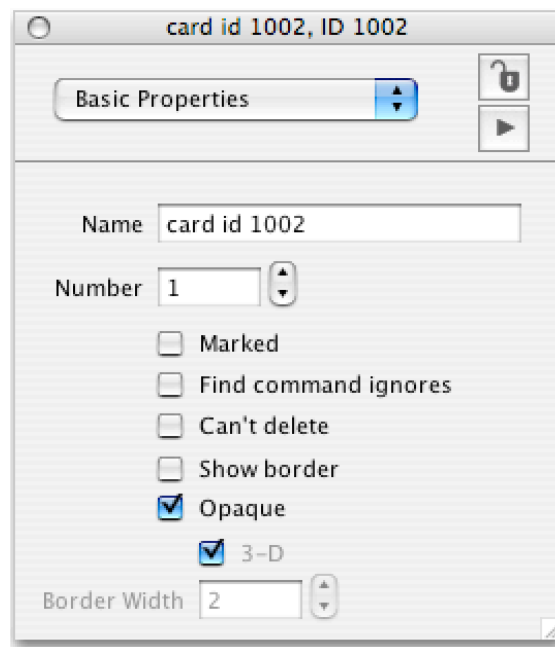


Fig. 23-1. Card Basic Properties.

NAME (*name*): If you do not name the card, its name, by default, is its *id* number. You can refer to the *id* number in a script such as:

go to card id 1002

You could also refer to a card by its card number in the stack. For instance, if you wanted to go to the third card in the stack you could write:

go to card 3 of this stack
or

go to the third card of this stack

However, it is generally best to give the card a name, rather than referring it by id number or by position in the stack, since a name is more recognizable when you read a script, and the card position in the stack can change. A name, though, remains constant. For example:

go to card "menu" of this stack

Important! Whenever assigning a name to any object in LiveCode, be sure to enclose the name in quotes. This is absolutely necessary if the name consists of more than one word. However, even if the name is a single word, in which case the script may work without quotes, it is still better to enclose the word in quotes, to eliminate incorrect functioning of the script in case the name coincidentally is the same as another key word in the LiveCode vocabulary.

NUMBER (*number*): The sequential number of the card. Note: If you add, delete, or reposition cards within the stack, this number may change.

MARKED (*mark*): Checking this marks the card, flagging it for reference in a script. For instance:

go to the next marked card

The word *mark* can be used as a property or as a command. E.g.:

set the mark of this card to true -- a property
mark this card -- a command

FIND COMMAND IGNORES (*dontSearch*): This refers to any fields on the card. Checking the box indicates that when searching for a string of text in the stack, the search bypasses the fields on that particular card.

CARD SIZE & POSITION (Fig. 23-2)

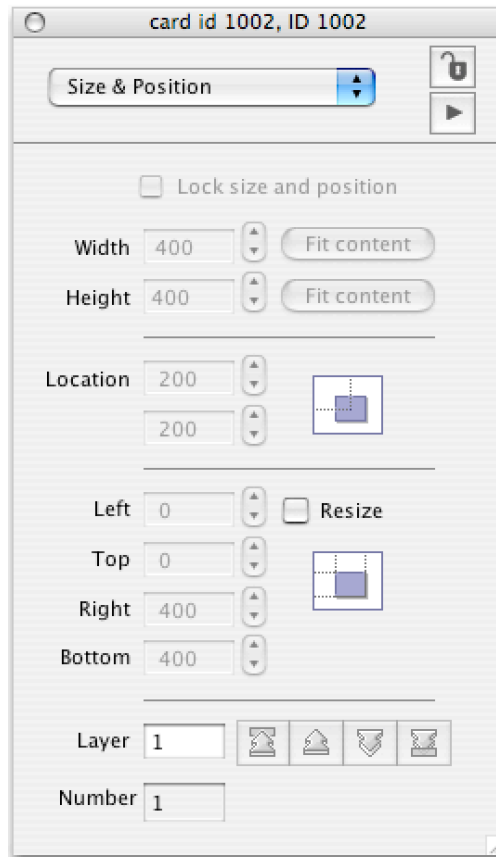


Fig. 23-2. Card Size and Position.

Note that you cannot set height, width, and location for individual cards. The height, width and location of a card are the same as that of the stack.

For cards, **Layer** (*layer*) and **Number** (*number*) are the same thing. **Layer** and **Number**, however, can be different for objects on a card. If you have a number of buttons and fields on a card, for instance, the Layer refers to the position of the control in reference to *all* the controls on the card, while the Number refers to the position of the control in reference to **other controls of like kind**. For instance, if you successively put 2 buttons on a card, and then put 3 fields on a card, the last field would have a **Layer** of 5, but a **Number** of 3. The arrows to the right of Layer and Number move the control closer or farther from the card.

When you refer to *button 1*, the reference is to the button's number, rather than its layer. A button that is called *button 1* could be far from the card surface in layer 3, for instance, if there are fields that are closer to the card than is the button.

Card Scripting

on closeCard

on openCard
on preopenCard
mark card
unmark card
lockmessages

on closeCard/ on openCard/ on preopencard

On closeCard indicates what should be done while the card is closing. *On openCard* indicates what should be done as soon as a card is open and visible. For instance, in a 2-card stack, if you have this script in card 1:

on closeCard
answer "I'm card 1 and I'm sorry to see you leave."
end closeCard

then the above message will appear, just before you leave card 1.

If you put the following in the script of card 2:

on openCard
answer "I'm card 2. Welcome."
end openCard

then the message will appear after card 2 is open.

PreopenCard, though, acts behind the scenes before the card is open. *PreopenCard* is useful, since you might want to do some shuffling around of objects on the card before the card is actually seen.

mark card/ unmark card

Marking a card either manually in the Basic Properties of a card's Property Inspector, or by script, flags it for future reference in a script. Examples:

mark this card
unmark this card
go to next marked card
get the number of marked cards -- tells you how many cards in the stack have been marked
mark cards where field "Customer name" contains "Arthur"

lockMessages/ lock messages

Locking messages enables you to navigate to other cards quickly by bypassing any *openCard*, *closeCard*, or *openStack* messages they may have. Examples:

set lockMessages to true -- or *lock messages*

When a handler is no longer executing, *lockMessages* automatically reverts to false.

CHAPTER 24. BUTTON PROPERTY INSPECTOR

BUTTON BASIC PROPERTIES

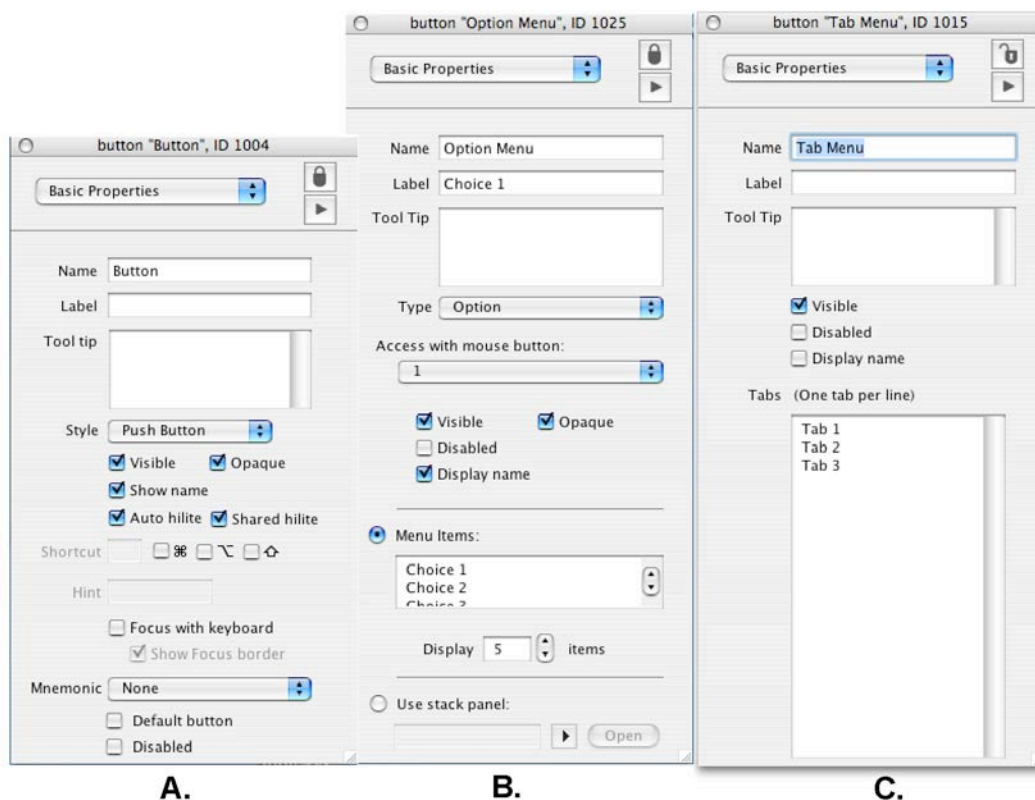


Fig. 24-1. Button Property Inspectors.

TOOL TIP (*toolTip*): Whatever you type here will show up as a small note when the user moves the end of the cursor over the button in Run Mode. It is a very useful way to provide the user with additional information when the cursor hovers over the button. It can be more than one line.

The tooltip can be used as an educational tool to quickly let the user see the script of the underlying control on hovering over the control, as in the following script:

```
on mouseEnter  
  set the tooltip of the target to the script of the target  
end mouseEnter
```

VISIBLE (*visible*): Indicates whether or not the button is visible.
Example script:

```
set the visible of btn 1 to false  
or  
hide btn 1
```

Remember, if the visible of a button is false it will not respond to a mouseclick. However, the button will respond if a message such as *dispatch mouseUp to button "MyButton"* is sent from another control. The button will also if the reason for the invisibility is that the button's blendlevel is 100.

SHOW NAME (*showName*): Indicates whether or not you want the button to show or not show its name (label).

AUTOHILITE (*autoHilite*): When checked, the button will highlight when the mouse is clicked down, to indicate that the button is being clicked on.

SHAREDHILITE (*sharedHilite*): If the button is in a group that behaves like a background on different cards and you want it to have the same hilite state on every card, check the **Shared Hilite** box (sets the *sharedHilite* property to true). If you want the same button to be able to show different hilite states on different cards, leave the **Shared Hilite** box unchecked. Typically, you want the *sharedHilite* of grouped checkboxes and radio buttons to be false, so that the user can select different hilite states on different cards.

DISABLED (*disabled*): Grays out the button, and renders it non-functional.

The **Basic Properties** section of the Property Inspector for **menu buttons** (option, pulldown, combo box, and pop-up menus) looks somewhat different, as follows (**Fig. 24-1B**):

MENU ITEMS: The place where you list the menu choices to be displayed on the menu button.

The **Basic Properties** section of the **Tab Menu** (Tab Panel) button (**Fig. 24-1C**), has a Tabs box for giving names to the individual tabs. The script of the Tab Menu button resembles that of other menu buttons. Commonly, the Tab Menu control is used to navigate to different cards or to show changing images or text in the box below the Tabs.

BUTTON ICONS & BORDER (Fig. 24-2)

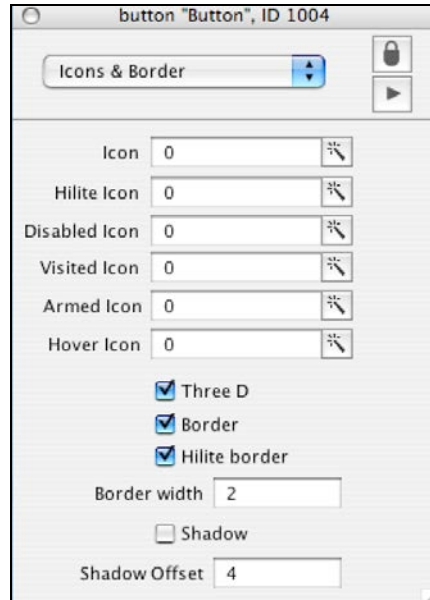


Fig. 24-2. Buttons Icons & Border.

ICON (*icon*): You can choose any image, even an animated GIF, to appear as an icon on the button by choosing its ID number, provided the image resides with the stack, whether visible or invisible. A good place to store the image is the mainstack, since all the substacks refer back to the mainstack.

The position of the icon (left, center, or right justified) on the button can be set in the **Text Formatting** section of the button object inspector.

HILITE ICON (*hiliteIcon*): With this option, a different icon of choice appears when the mouse is down on the button, returning to the original icon when the mouse is released (i.e., the mouse is up) or the cursor moves outside the button.

DISABLED ICON (*disabledIcon*): Changes to a “disabled” icon of your choice when the button is disabled. E.g.:

set the disabledIcon of btn 1 to 210001

HOVER ICON (*hoverIcon*): Specifies your icon of choice when the mouse hovers over the button.

SHADOW (*shadow*): Determines if the button has a shadow.

BUTTON COLORS & PATTERNS (Fig. 24-3)

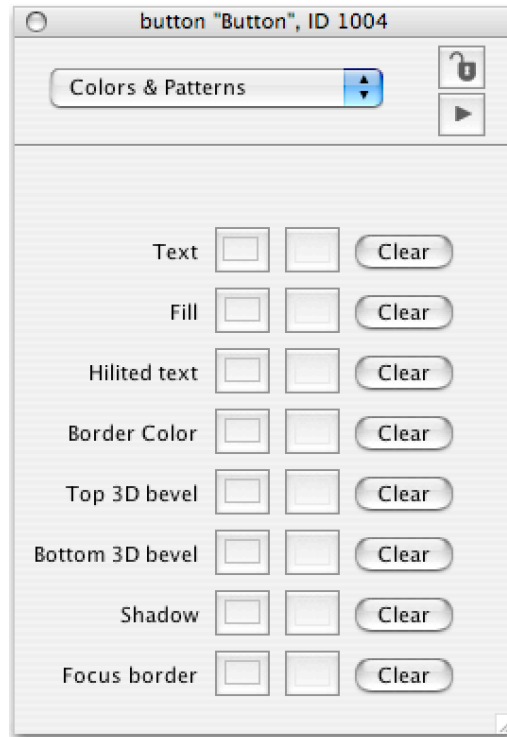


Fig. 24-3. Button Colors & Patterns.

TEXT (*foregroundColor*, alternatively, *textcolor*): Sets the button's text color

FILL (*backgroundColor*; *backgroundPattern*): Sets the button's background color or pattern.

HILITED TEXT (*hiliteColor*; *hilitePattern*): Sets the color or pattern *surrounding* the button's text when the button is clicked on. To see this effect, uncheck the **Three D** box in the **Icons & Border** section of the Property Inspector.

BORDER COLOR (*borderColor*; *borderPattern*): Colors the button's border. To see this effect, uncheck the **Three D** box in the **Icons & Border** section of the Property Inspector, and be sure the button's border is set wide enough to see. You can do the same for a card or field.

SHADOW (*shadowColor*; *shadowPattern*): Sets the color (or pattern) of the button's shadow. Be sure the **Shadow** box is checked in the **Icons & Border** section of the Property Inspector.

BUTTON GRAPHIC EFFECTS (Fig. 24-4)

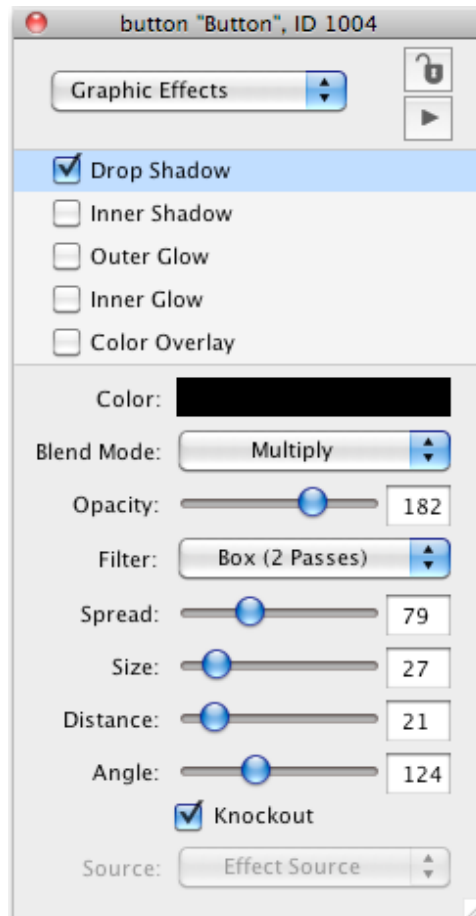


Fig. 24-4. Button Graphic Effects.

Button graphic effects allow for changes in the appearance of buttons through alteration of a number of features of shadows and glow, and other graphic variables. Perhaps the best way to learn about these is to explore the various sliders and checkboxes in this part of the button Property Inspector.

BUTTON BLENDING (Fig. 24-5)

BLEND LEVEL (*blendLevel*): Note how sliding the **Blend Level** Bar alters the transparency of the button. You can make the button partially or fully transparent if you wish. The button's script works on clicking with the mouse even if the button is fully transparent.

INK (*ink*): The best way to demonstrate blending is by trying out the various ink options, which will alter the coloring of the object you are working with, particularly when overlying another object. The effects are much more dramatic when applied to a color image. You may wish to alter the blend for artistic reasons, or, when using an image as a button, alter the hilite of the image through scripting when the mouse is applied to it, to indicate when the mouse has entered, left, or is pressed down on the image. E.g. :

```
on mouseDown
    set the ink of btn 1 to "notSrcCopy"
end mouseDown
```

```
on mouseUp
    set the ink of btn 1 to "srcCopy"
end mouseUp
```

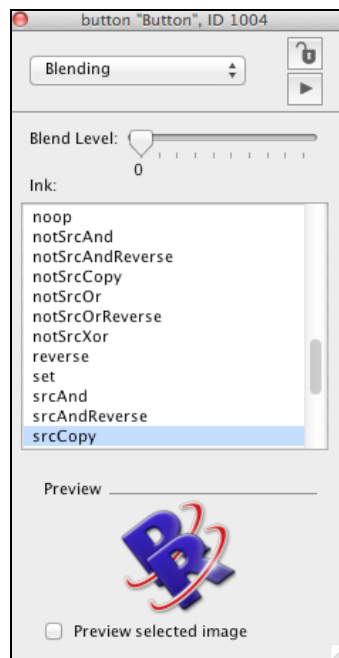


Fig. 24-5. Button Blending.

BUTTON SIZE AND POSITION

The **Lock Size and Position** checkbox locks the position and size of the button, so that it does not move around or resize inadvertently during development. When the box is checked, the button cannot be moved with the mouse. However, it can still be moved with the arrow keys when selected in Edit mode, more quickly (10 pixels vs 1 pixel) if you hold the Shift Key down while using the arrow keys.

FIT CONTENT: Narrows the button to fit the length and height of the words.

LOCATION (*loc*): The *loc* of a button is where the center of the button lies, in x and y coordinates relative to the **stack**. The upper left corner of a stack by default has a loc of 0,0. The diagram to the right of the **Location** area illustrates the x and y axes of the button's loc (**Fig. 24-6**).

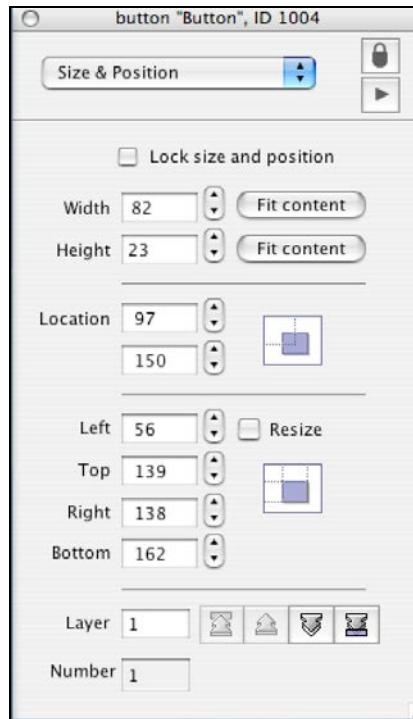


Fig. 24-6. Button Size & Position.

LEFT (*left*), **RIGHT** (*right*), **TOP** (*top*), **BOTTOM** (*bottom*): Sets the locations of the left, right, top, and bottom of the button. **Left** and **Right** are x coordinates, while **Top** and **Bottom** are y coordinates. Together, all four constitute the *rect* of the button. E.g.,

```
set the left of button 1 to 225
set the top of button 1 to 256
set the right of button 1 to 339
set the bottom of button 1 to 326
```

The above four lines of code are the equivalent of writing:

```
set the rect of button 1 to 225,256,339,326
```

LAYER (*layer*): Indicates the layered position of the button in reference to all other controls on the card.

NUMBER (*number*): Indicates the position of the button in reference only to other buttons on the card. Thus, a button's **Number** may be less than a button's **Layer**.

BUTTON TEXT FORMATTING

MARGINS: Sets how far left or right justified text can be set next to the margin of the button. You can add an icon and make it appear either to the left, right, or top

of the text depending on whether or not you choose left, center, or right justified in the button's text formatting.

CHAPTER 25. MENU PROPERTY INSPECTOR (Fig. 24-1C)

Menu controls (**option menu, pulldown menu, combo box, and pop-up menu**) may look like fields, but are not, even though they may have lists of words in their menus. They are called **buttons**, but have a special scripting that deals with their lists of words:

menuPick

Say an option menu, pulldown, menu, combo box or pop-up menu control on a card has options in its list for:

choice 1
choice 2
choice 3

The button script might read something like:

```
on menuPick pItemName
  switch pItemName
    case "Choice 1"
      answer "You picked a good one."
      break
    case "Choice 2"
      answer "Not such a good choice."
      break
    case "Choice 3"
      answer "That is even worse than choice 2."
      break
  end switch
end menuPick
```

on menuPick means "When you pick an item from this menu".

The *switch pItemName* part is like saying "Select from the following options based on the value of *pItemName*".

Case "Choice 1" means "In the case where the *pItemName* you selected is "Choice 1".

Break signifies to Livecode that you no longer wish to carry out any further instructions for this case.

end switch, like *end if*, ends the entire *switch* script.

An alternative menu scripting resembles the *if/then* structures of other scripting situations is:

```
on menuPick pItemName
  if pItemName is "choice 1" then
    answer "You picked a good one."
  end if
  if pItemName is "choice 2" then
    answer "Not such a good choice."
  end if
  if pItemName is "choice 3" then
    answer "That is even worse than choice 2."
  end if
end menuPick
```

You could also accomplish the same functionality, however, with *if/then* statements if you want.

When dealing with a significant number of menu items, many people find it easier to use the *switch/ case/ break* format, but you can stick with the format you feel most comfortable with. The *if-then* format may be more attractive to others (like me), because it is more English-like and resembles LiveCode's standard *if-then* scripting.

LiveCode's **Menu Builder** will not be discussed here. For most purposes it will suffice to create your own menu using LiveCode's menu controls. A nice menu can be created by preparing a group of Pulldown Menus aligned side to side. Subcategories of each menu item can be prepared simply by using Tab indents on the choices list in the menu button Property Inspector.

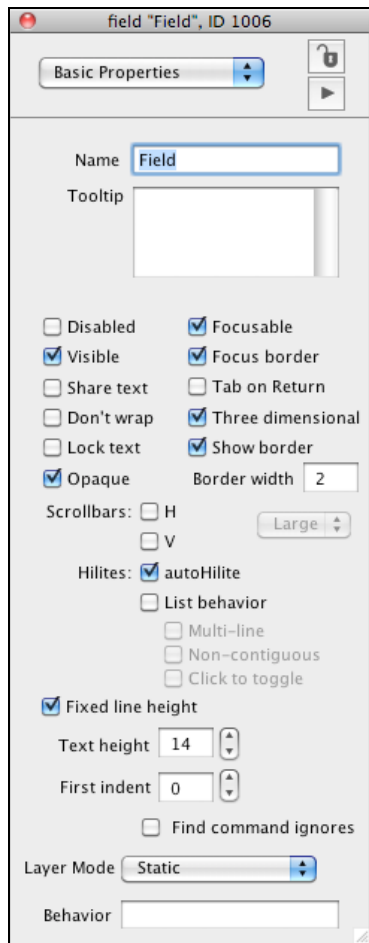
You can also create an interesting type of menu by creating your own tools palette in the form of a palette stack that is called upon to open from the card.

Rather than a floating pallet stack separate from the card, you can have the stack appear as a pulldown menu from a pulldown menu button. You do this by checking the radio button "Use Stack Panel" at the bottom of the Basic Properties section of the pulldown menu button's Property Inspector.

CHAPTER 26. THE FIELD PROPERTY INSPECTOR

Close LiveCode if it is not already closed (no need to save), and then reopen it.

FIELD BASIC PROPERTIES (Fig. 26-1)



26-1. Field Basic Properties.

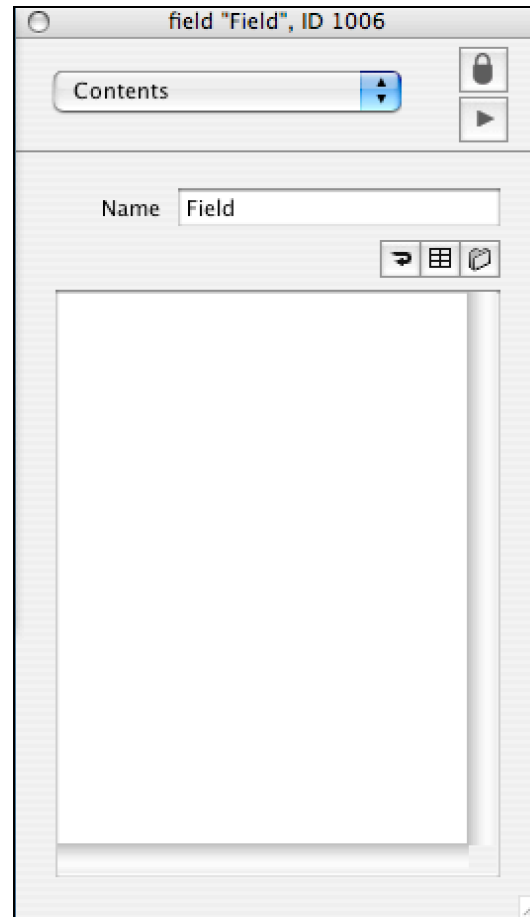


Fig. 26-2. Field Contents.

Create a new main stack, place a text entry field on its card and open its Property Inspector. You'll know it's a field Property Inspector because it says so on the field's Property Inspector title bar, which also indicates the field's ID number.

NAME (*name*): You can refer to a field in a script by number or its ID, but it is recommended that you refer to fields (and all other LiveCode objects) by name, though, since names are more descriptive and easier to recognize in a script than ID numbers. Also, the layered position of a field may change with development, and its ID number will change with copying and pasting the field, thereby invalidating the script reference to a particular field number or ID.

As with other objects in LiveCode, when referring to a name in a script, put it in quotation marks, so the name is not confused with other LiveCode words or variables you have defined. Also, a name enclosed within quotes can be more than one word. Without the quotes, LiveCode does not recognize a name with more than one word in it.

set the name of field 1 to Score -- works

set the name of field 1 to "My Score" -- works
set the name of field 1 to My Score -- doesn't work (no quotes)

Note the difference between using *name* and *the short name* in a script:

1. Type the words "Goodby cruel world" in field 1
2. Name field 1 **Goodbye**
3. Now, open the Message Box and compare the following scripts typed in the Message Box:

put the name of field 1 -- returns field "Goodbye" in the Message Box
put the short name of field 1 -- returns Goodbye in the Message Box
put field 1 -- returns Goodbye cruel world (the contents of field 1) in the Message Box.

SHARE TEXT (*sharedText*): If you make the field into a group that is placed on multiple cards with **Behave Like a Background** checked (set to *true*), do you want the text you type into one field to automatically be the same on all the cards or not? For instance, if every card contains a field at the top containing the name of your company, you would want to check the **Share Text** option. However, if you want the field to contain different information for each card it is on (e.g. the name of each new client), you would want to leave the **Share Text** box unchecked.

DON'T WRAP (*dontWrap*): If you check this box, the line you type will continue going beyond the right edge of the field until you type a Return to start a new paragraph. If you leave it unchecked, the line will automatically stop at the right edge of the field and continue to the next line without starting a new paragraph.

LOCK TEXT (*lockText*) – This is a critically important feature of the field Property Inspector. If **Lock Text** is left unchecked, the user will be able to type into the field. If the text is locked, the user cannot type in the field, but may still be able to select (see **Focusable** below). When locked, the field will behave like a button, responding to mouse clicks to do things, according to the field's script.

OPAQUE (*opaque*). When unchecked, the space around the letters is transparent, and the underlying color and texture of the card or other object shows through. When checked, the color of the background of the letters is by default white, but could be any color you wish to set it at in the **Colors & Patterns** section of the field's Property Inspector.

FOCUSABLE (*traversalOn*): When checked, it means that the user can select and copy the text within the field, even if the field is locked and the user cannot type text in the field.

FOCUS BORDER (*showFocusBorder*): When this is checked (the *showFocusBorder* is true), the border of the field will hilite when the user clicks on the field, regardless of whether or not the field is locked. Hiliting a border

confirms to the user that the field has been clicked on.

TAB ON RETURN (*autoTab*): When Tab On Return is checked, the cursor, when at the bottom of a field, moves to the next field number when the Tab, Return, or Enter key is pressed, provided the field is unlocked and its **Focusable** (*traversalOn*) quality is checked.

SCROLLBARS (*hScrollBar*; *vScrollBar*): Scrollbars H and V allow you to add a horizontal or vertical scrollbar to the field, which help when the text extends beyond the boundary of the field.

AUTOHILITE (*autoHilite*): When checked, the text that the user selects within a field is highlighted, as it would be with a highlighter pen, provided the field's *traversalOn* (**Focusable**) is also on.

LIST BEHAVIOR has the subcategories, **Multiline**, **Non-contiguous**, and **Click to Toggle**. When **List Behavior** is checked but none of the latter three are checked, the line the user clicks on in a locked field is hilited and selected as a whole, as it does in any list field, but the user cannot hilite more than one line at a time. This feature is useful in that the script for the field may have directions to do something when the line as a whole is clicked. When **Multiline** is selected, the user can hilite more than one line by either dragging across the lines or selecting one line with the Shift Key and then selecting another with the Shift Key still down.

By also checking **Non-contiguous**, you can select non-contiguous lines by Command-clicking (Mac) or Control-clicking (Windows and Unix). So you can independently select, say, the first and 4th lines without those in between.

When **Click to toggle** is checked, clicking on a line alternately selects or deselects it.

FIXED LINE HEIGHT (*fixedLineHeight*): Say you have five lines in the field and one of the words on line 2 is superlarge. The word might overlap line 1 and look bad. If you uncheck **Fixed Line Height**, the distance between lines 2 and 1 will increase to adjust for the increased height of the word. Otherwise, the lines will remain the same distance from one another. Another way of controlling this problem of large words is through the **Text Height** (*textHeight*) option. Increasing the **Text Height** will in general increase the spacing between all the lines.

TEXT HEIGHT (*textHeight*): Sets the distance between lines. Also sets the row width in tables.

FIRST INDENT (*firstIndent*): The default is 0, meaning that there is no paragraph indentation. If you increase the **First Indent**, the first word of each paragraph will be indented accordingly. Conversely, if you make the **First Indent** a minus number, the first paragraph line will not be indented, but the other lines will! This

is great, but may cause the first word of the paragraph to be situated to the left of the left margin of the field, and not visible. This can be corrected by increasing the **Margins** property of the field, which can be done in the **Text Formatting** section of the Property Inspector (discussed below).

FIND COMMAND IGNORES (*dontSearch*): A very useful feature! There is a *Find* command that can find any words wherever they may be in any field in the stack. For instance, you may have an invoice stack with a different customer address on each card and vaguely remember that one of the customers is from California. You might type in the Message Box:

Find “California”

to find the card that has California on it. However, what if your own company is in California and has “California” on every card as part of the company address field. The search would be useless, since you would find “California” on every card. However, if you checked **Find Command Ignores** in your company’s address field, LiveCode knows to search all the fields, except the one with your company address.

FIELD CONTENTS (Fig. 26-2)

The **Contents** box provides an alternate way to type in the contents that you want to show in the field, apart from typing in the contents directly into the field. This can be useful when the field is locked and you don’t want to have to unlock it to alter its contents, forgetting that you unlocked the field.

The Return icon button (curved arrow) in the **Contents** section of the field Property Inspector toggles the text in the **Contents** section to wrap or not wrap around when it reaches the right edge of the **Contents** box. It has no effect on what the user actually sees in the field itself. To effect what the user sees regarding wrapping in the field, check or uncheck the **Don’t Wrap** box in the **Basic Properties** section of the field Property Inspector. Lists fields are always set to **Don’t Wrap**, even if you uncheck the **Don’t Wrap** box.

To transfer text from another document (e.g. Word), simply select the text and slide it onto the card field.

FIELD COLORS & PATTERNS

TEXT (*foregroundColor*, *foregroundPattern*): Enables you to set the color or texture of the text. (An alternative word is *textcolor*, which is easier to remember.

FILL (*backgroundColor*, *backgroundPattern*): Fills in the background behind the text.

You can be even more specific than setting the field's entire background color; you can change the background color of individual lines or words in a field, such as in the script:

set the backgroundColor of line 3 of field 1 to "green"
set the backgroundColor of word 1 of line 2 of field 1 to red

TEXTHILITE (*hiliteColor*, *hilitePattern*): Sets the color of the background around the text when the text is selected (like a highlighting marker pen would do).

SHADOW (*shadowColor*, *shadowPattern*): Sets the shadow color or pattern of the field. You first have to give the field a shadow and a shadow size (*shadowOffset*) through scripting, as in:

set the shadow of field 1 to true -- gives the field a shadow
set the shadowOffset of field 1 to 10 -- sets a size to the shadow
set the shadowColor of field 1 to "blue"

FIELD GRAPHIC EFFECTS

Experiment with these graphic modifications of shadows. The shadow effects also apply to text when the field is transparent.

FIELD SIZE AND POSITION (Similar to Button size and position)

FIT CONTENT: Clicking on **Fit Content** resets the width and height of the field to fit the field's content.

FIELD TEXT FORMATTING (Fig. 26-3)

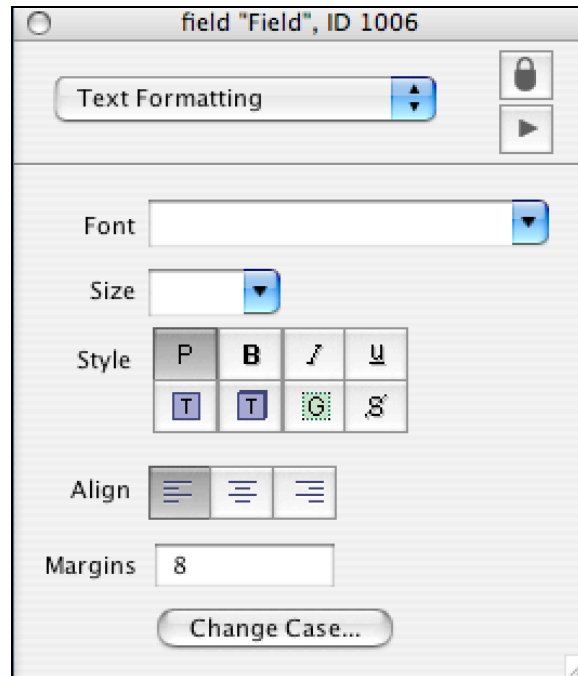


Fig. 26-3. field Text Formatting.

FONT (*textFont*): Sets the type of font for the field. If the field's **Font** option is empty, the default is that of the card. If the card's **Font** field is empty, the default is the **Font** setting of the stack. If the stack's **Font** field is also empty, the default is that of the system.

Sometimes a font that appears one way on Macintosh appears significantly different on Windows. I like to use Arial as a font that looks fairly the same on both computers, but it is a good idea to compare how your standalone application looks on both Mac and Windows.

SIZE (*textSize*): Sets the size of the text.

STYLE (*font*): Sets whether the font is **Plain**, **Bold**, **Italic**, or **Underline**.

MARGINS (*margins*): Sets the margins between the text and the edges of the field.

The **Change Case** button is grayed out in buttons; it applies to fields, though. In fields, you have the option of making the text all capitals, all lower case, the first letter of each word capitalized (Title case), or placing the first letter of sentences in capitals (Sentence case -- although this doesn't capitalize the first letter of the second and further paragraphs).

THE BASIC TABLE FIELD PROPERTY INSPECTOR

Maximum editable columns in the **Table** section sets the number of columns that can be typed in. **Tab Stops** in the **Table** section, sets the horizontal distance between cells in pixels, so a higher tab stop number makes a wider cell. **Cell height** is modified by the **Text height** field in the Basic Properties section.

Field Scripting

find

The basic *find* command finds a string of text in a field that might be anywhere in a stack. There are several forms of the command:

find (or *find normal*)
find chars
find string
find word (or *find words*)
find whole

Examples:

find (or *find normal*) -- finds the word(s) within any card field in the stack, even if parts of the word phrase are in different fields on a card, and in any order. If a card has a field with the word “leader” and another field on that card has the word “ring”, then *find “ring leader”* will find both these instances. It will also find the text if the words on the cards are “ringbearer” and “leadership” since it considers the first part of those words as part of the search. However, *find “ring”* will not find the word “boring”, since the “ring” is not at the beginning of the word.

find chars – does the same as *find normal*, but the characters can be in any part of the word. E.g. *find chars “ring leader”* will find a card containing “boring” in one field and “leader” in a different field.

find string – looks for a specific string that must be in the exact sequence as specified. E.g., *find string “leadership ring”* will find “leadership rings”, not “boring leadership” or even “ring leadership”.

find words – looks for complete words only. Thus *find words “leadership ring”* will find “leadership ring” or “ring leadership”, in any order, even separated on different fields of the card, but not “leaderships ring”.

find whole – This form is very specific. The words must not only be the whole words, but must be in the same sequence. Thus, *find whole “leadership ring”* finds “leadership ring” but not “leadership of the ring”.

find "Waldo" in field "countries" – tells you to search only in field "countries". Specifying the field name can save considerable time in the search process when there are fields in the stack with different names.

result

If the search is unsuccessful, LiveCode returns the message **not found** and places it in a container called the "result". The phrase for an unsuccessful search is "result is not empty". In other words, if the search were successful, there would be no complaints coming out of LiveCode; the result would be "empty". However, if the search is unsuccessful, the result is "not empty" because LiveCode returns the complaint of "not found". An example of this in a script would be:

```
on mouseUp
  find "leadership"
  if the result is not empty then beep – indicating an unsuccessful search
end mouseUp
```

There are many commands within Livecode that return their success or failure by placing information into *the result*. For example, if there is no card "Rumplestiltskin" there would be a beep with the following script:

```
on mouseUp
  go to card "Rumplestiltskin"
  if the result is not empty then beep
end mouseUp
```

char ***word*** ***line*** ***text***

LiveCode is extremely powerful in dealing with individual components of text, whether characters (chars), words, strings, or items, all of which are termed **chunks**. Example:

This is line 1 of this field.
It's a very nice line.
I think I'll make more lines like this.

put char 1 of word 4 of line 3 of field "MyField" -- Message Box reads **m**

put char 2 to 4 of word 4 of line 2 of field "MyField" -- Message Box reads **ice**

Spaces and returns are chars.

put line 3 of field "MyField" -- Message box reads **I think I'll make more lines like this.**

You can refer to the entire text of a field:

put the text of field "MyField" into message

or

the text of field "MyField" – same thing as the above script line, but abbreviated.

or

put field "MyField" – even more abbreviated

Note that the script word *line* refers to a string of characters up to a Return (or Enter). If the line goes to the right end of the field and automatically wraps to the next line, it is still only one line.

the number of lines in field "test" – returns to the Message Box the number of lines in the field.

Item

itemDelimiter

By default, an item is any string of characters, word or words separated from others by a comma. For example, consider a field titled "MyField" with the following text:

Apple sauce, sun dried tomatoes, corn on the cob (or off the cob)

there are 3 items: **apple sauce**, **sundried tomatoes**, and **corn on the cob (or off the cob)**. *Item 1* of the field is **apple sauce**.

The item number can also be phrased as a negative, For instance,

item 2 of line 1 of field "MyField" returns **sundried tomatoes**.

The itemDelimiter does not have to be a comma. For instance, consider the following script:

on mouseUp

set the itemDelimiter to "(" -- switches the default *itemDelimiter* to "(" rather than a comma

put item 1 of field "MyField"

end mouseUp

The Message Box will then show "Apple sauce, sun dried tomatoes, corn on the cob", because there are now only 2 items in the list, separated by the "(" rather than a comma.

Variations on the usage of *item*:

put **first** *item of line 1 of field "test"*
put **last** *item of line 1 of field "test"*

The *itemDelimiter* can be any character you want. Once the script handler ends, you don't have to go to the trouble of setting the *itemDelimiter* back to a comma. It automatically returns to the default comma on termination of the handler.

However, in long handlers it is good practice to return to the default delimiter (the comma) mid-handler once you are finished with what you want to do with your special delimiter, in case you do other things in the handler that require the default.

A **word** cannot contain items. Therefore:

item 2 of word "a,b,c" -- returns an error message
item 2 of "a,b,c" -- returns "b"

before
after
into

Back to the same text:

Apple sauce, sun dried tomatoes, corn on the cob (or off the cob)

Now type into the message box:

put "Bread, " before field "MyField"

This inserts the word "Bread, " at the beginning of the line so that the line reads **Bread, Apple sauce, sun dried tomatoes, corn on the cob (or off the cob)**

put ", wine" after field "MyField" would insert **,wine** as the last item listed.

put "Let's go home." into field "MyField" would replace everything in the field with the words "Let's go home." When you put something *into* something else in LiveCode, whether it is a field or a variable, everything in that field (or variable) is replaced by what you put in.

put "3000" into tAmount -- puts 3000 into the variable *tAmount*, replacing everything that is in *tAmount*.

put “Harold” & return & “Kate” into field “Names” -- this inserts 2 lines, one saying **Harold** and the other **Kate** into a field you had named “Names”. The *return* part of the script inserts a carriage return, thereby putting Kate on a new line.

begins with
ends with

These phrases specify a position. For example, a script line that reads:

if “Let’s go home.” begins with “Let’s” then beep

there will be a beep sound, since “Let’s go home.” does begin with “Let’s”.

if “Let’s go home.” ends with “home” then beep

This results in no beep, since “Let’s go home does not end with “home” but with “home.” (includes a period).

clickLine [compare with “line”]
value
clickText
link
foundText
foundLine
foundField

Imagine you have a card with a single field named **Animal**, with a text that reads:

Dinosaurs can be big or small.
They evolved into birds.

When you search for “Dino”:

- the *foundChunk* = **char 1 to 9 of field 1**
- the *foundText* = **Dinosaurs**
- the *foundLine* = **line 1 of field 1**
- the *value of the foundLine* = **Dinosaurs can be big or small.**
- the *foundField* = **field 1**
- the *value of the foundField* = **<all the text in the field>**
- the *name of the foundField* = **field “Animal”**
- the *short name of the foundField* = **Animal**

on linkClicked myText

An *on linkClicked myText* handler means “When the text on which you have clicked is a linked word or words”. *MyText* is a made-up temporary container signifying “whatever the linked group is that you clicked on”. Example:

```
on linkClicked MyText  
  go to card MyText  
end linkClicked
```

If the text you clicked on in a field is a link (e.g. a pair of linked words in a list, such as **Florida birds**, the script directs navigation to go to the card of the same name (**Florida birds**) as the link.

select
selectedLine
selectedText
selection
select before/ select after
put empty
replace

You can regard *the selection* as a container into which you can add words or numbers. Thus:

put “something else” into the selection -- completely replaces your selection with “something else”, even if your selection is just an insertion point.

put empty into the selection -- putting *empty* deletes the selection. You can do this with an entire field or a selection within the field:

put empty into word 2 of field 1 -- just removes word 2
put empty into field 1 -- removes the text in the entire field

An alternative to *put empty* is *put “”*.

The word *none* is equivalent to saying “0”. So while putting *empty* into a field erases the field, putting *none* into the field would put a “0” into the field.

replace

Replace replaces every instance of a word or phrase in a field or other container with another. E.g.,

replace “which” with “that” in field 1

You can do this for an entire stack at once with **Find and Replace...** in the **Edit** menu.

scroll

The *scroll* (or *vscroll*) is the amount a field or group is scrolled down in pixels. For instance:

set the scroll of field "MyField" to 50

The field text will shift vertically, setting the scroll to 50.

on enterInField/ on returnInField

On enterInField and *on returnInField*, when written as handlers in an unlocked field's script, tells LiveCode what to do when the Enter key or Return key is pressed while typing in the field. *ReturnInField* and *enterInField* are very useful when you have a series of fields in which you want to successively move from one field to the next after pressing Return or Enter.

on returnInField -- this script would be in field 1

select after text of field 2 -- puts an insertion point after the text in field 2 after the return key is pressed.

end returnInField

CHAPTER 27. THE IMAGE PROPERTY INSPECTOR

There are two ways to bring an image into your stack. The first is direct, importing it as a control that is incorporated into the stack's memory. The other way is to keep the image outside the stack and to "reference" it from within an **Image Area** Property Inspector.

The direct import way has the advantage in that you don't have to keep track of where the image is outside the stack. It is always a part of the stack. The disadvantage is that it takes up a certain amount of memory when the stack is loaded. The greater the number of imported images, the greater the memory requirements of the stack and the longer it takes for the stack to load, since all directly imported images are loaded at once.

IMAGE BASIC PROPERTIES (Fig. 27-1)

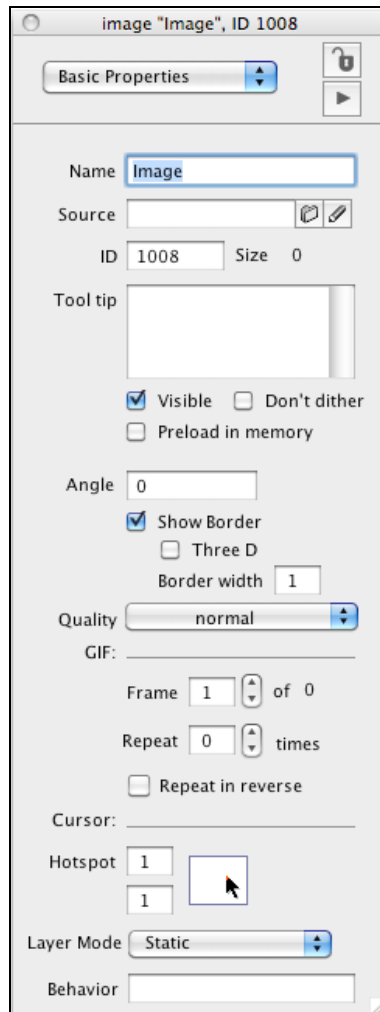


Fig. 27-1. Image Basic Properties.

To import images as a referenced control, either:

A. create an **Image Area** object through the LiveCode menu bar (**File/ New Referenced Control/ Image File...**)

or

B. move an **Image Area** object from the Tools Palette onto the card. Open the **Image Area** Property Inspector to **Basic Properties** and select the **Source** folder icon. It will prompt you to browse to find your chosen picture of reference.

The Property Inspector for an image is the same whether or not the image is directly imported or referenced. The **Source** field, however, will of course be empty when the image is directly imported, since the image resides in the stack rather than at a distant location.

Next to the source folder icon in the image Property Inspector is a **pencil icon**. This is useful in that it connects you with your image editor (e.g., Photoshop) to make on-the-spot alterations to your image. You can also reach your image

editor by Right-clicking on the image (Control-click with a one-button mouse) and selecting **Launch Editor** from the pop-up menu. The editor used is the image editor specified in the **General** panel of the LiveCode preferences. If an image editor has not been specified, you will be asked if you wish to specify one now.

When you use an image editor, such as Adobe Photoshop, to alter a referenced image, the actual image that is stored outside of LiveCode is changed as well. However, if you use the image editor to alter an image that is directly imported as a control, the original image outside of LiveCode does not change.

When right-clicking on a directly imported image control (Control-clicking with a one-button mouse), one of the options from the pulldown menu is **Magnify**. This produces a small pulsating square in the center of the picture and a separate window with a magnification of the pulsating square area. The new window can be modified with the paint tools. The edges of the pulsating square can be grasped and moved, to magnify different areas of the image. This works only on directly imported images, not referenced images.

ID (*id*): An image can be referred to by its ID number, but it is easier to interpret a script when an image is referred to by the name you give it than by a non-descriptive ID number.

To the right of the **ID**, the word **Size** indicates the amount of memory, in bytes, that the image takes up when the stack is opened. For referenced images, the **Size** is 0, since the image does not reside in the stack.

ANGLE (*angle*): Rotates the image (but not referenced ones). Even an animated GIF can viewed at a rotated angle.

GIFs, including animated ones, can be brought in with their alpha channels, so they can exhibit transparencies.

FRAME (*currentFrame*): Sets the current frame number of the animated GIF. If you move back and forth between cards, though, LiveCode doesn't remember the frame information, so you have to reset it in a script when the card opens. For example:

```
on preopenCard
  set the currentFrame of image "Rotating Dollar" to 2
end preopenCard
```

REPEAT (*repeatCount*): Sets the number of times the animated GIF will repeat when the card opens. If you don't want the GIF to play at all when the card opens, set the **Repeat** (*repeatCount*) to 0. If you want it to play continuously, set the *repeatCount* to -1.

REPEAT IN REVERSE (*palindromeFrames*): When *palindromeFrames* is set to

true (**Repeat in Reverse** checked), the GIF loops back and forth. When *palindromeFrames* is false, a repeating GIF skips back to the beginning to start a new loop.

HOTSPOT (*hotSpot*) : Should you wish to use an image as a cursor, you may want to specify the point on the image that is active (the *hotSpot*) in sending the mouse message. For instance, a *hotSpot* of 1,1 (the usual default) will set the hot spot at 1 pixel from the left side of the image and 1 pixel from the top of the image (the upper left corner of the image).

IMAGE GRAPHIC EFFECTS

Image Graphic Effects enables you to create and modify shadows around images. Playing around with these features will give the best idea of how they can be used.

IMAGE SIZE & POSITION

The **Lock Size and Position** checkbox locks the position and size of the image, so that it does not move around or resize inadvertently during development. When the box is checked, the image cannot be moved with the mouse. However, it can still be moved with the arrow keys when selected in Edit mode.

The **Lock Size and Position** checkbox is **particularly important** in dealing with imported images and movies. If you try to resize a picture or movie on a card and then go back and forth from another card, you will find to your consternation that the picture and movie resize to their original dimensions. However, this is prevented when **Lock Size and Position** is checked.

CHAPTER 28. THE QUICKTIME PLAYER PROPERTY INSPECTOR

This information is mainly relevant to people using LiveCode 5.5.3 or earlier, with Quicktime 7.0 or earlier, which still can be used to create movies and QTVRs using the Quicktime player. Quicktime movies and Quicktime VR, though, are currently undergoing changes by Apple and LiveCode, so this information may need to be updated in the near future for later versions of Quicktime and LiveCode.

The Player object can reference Quicktime movies, MPEGs, Quicktime panoramas, animated GIFs, and even PDFs. It can also reference sounds, commonly in AIF or WAV formats.

PLAYER BASIC PROPERTIES (Fig. 28-1)

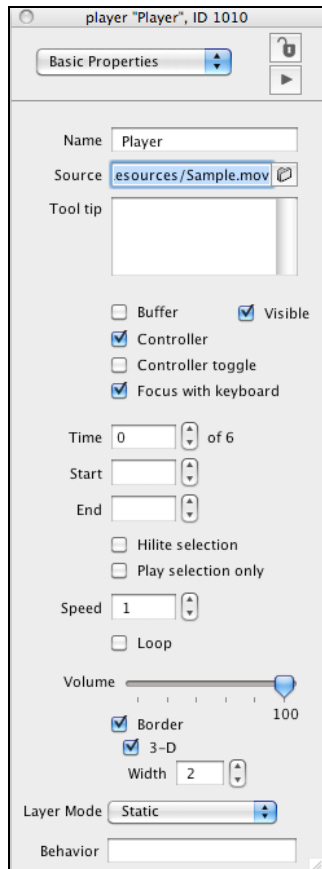


Fig. 28-1. Player Basic Properties.

BUFFER (*alwaysBuffer*): When **Buffer** is unchecked, the player window floats above all other objects on the card, so you can't, for instance, paste a button on top of the player; but the movie takes up less memory. When checked (*alwaysBuffer* is true), the movie takes up more memory when playing and the controller doesn't work. However, you can paste buttons or other objects on top of the movie, and, despite the lack of a functioning controller, you can control the movie's play features through scripting.

CONTROLLER (*showController*): Gives you the option of showing or not showing a controller.

FOCUS WITH KEYBOARD (*traversalOn*): When true (checked), allows users to affect the movie with standard Quicktime keyboard shortcuts, e.g. spacebar to start and stop playback.

TIME (*currentTime*): While a movie may be only a few seconds long, its *duration* is much longer, because duration is (the number of seconds in the movie) x the number of intervals per second in the movie).

Since *duration* is so much larger than *seconds*, one uses *duration* units rather than *seconds* to precisely determine times within a movie.

The *currentTime* describes in *duration* units the particular time where the movie is at. E.g.:

put the currentTime of player "myMovie" into msg

Look at the **Time** (*currentTime*) field for your own information about where the movie is in time. For instance, if you want parts of a movie to start or stop playing at particular points in time, you first want to know what those times should be, to create the **Start** or **End** times. In that case:

- a. Use the controller to find the points in the movie that you would like to make your **Start** (*startTime*) and **End** (*endTime*) times. The **Time** (*currentTime*) field will show you those times.
- b. Put this information in the **Start** or **End** time fields.

Script example:

set the startTime of player "My movie" to 200
set the endTime of player "MyMovie" to 300

However, in order for the **End** (and **Start**) features to work, you must first check the **Play Selection Only** (*playSelection*) box. Otherwise, the entire movie will play.

If you want some event to occur at some particular time within the movie, move the controller slider to that point and note the *currentTime*, which can then be used in a script or in a **Callback** (see below).

HILITE SELECTION (*showSelection*): The default is unchecked (*showSelection* is *false*). When checked (and **Play Selection Only** is also checked), users can set their own start and end times for the movie by Shift-dragging the player's thumb to see the start and end times, as indicated by a dark gray region within the controller.

SPEED (*playRate*): Sets the speed at which you want the movie to play. For this to work, the actual playing of the movie should be effected through a script, rather than through the player controller (i.e., if you only set the **Speed** through the player controller, the movie will still play at its original default speed. If you want to see the increased speed, you need a script like:

set the playRate of player "myMovie" to 2
play player "myMovie"

The default setting for **Speed** is 1. If **Speed** is set higher, the movie plays faster. If set to 0, the movie doesn't play. If set to a negative number, the movie plays backward.

LOOP (*looping*): When this is checked (*looping* is true), the movie will replay continuously, restarting at the beginning.

VOLUME (*playLoudness*): sets the volume of the Quicktime player. This can be done through the sound icon on the left side of the player controller, or set directly within the player Property Inspector through the volume slider. Or, the loudness can be set via a script:

set the playLoudness to 50 -- sets the general speaker volume for all sounds.

set the playLoudness of player "MyMovie" to 50 -- sets the volume of the specific player only, as a percentage of the general *playLoudness*

PLAYER TRACKS (Fig. 28-2)

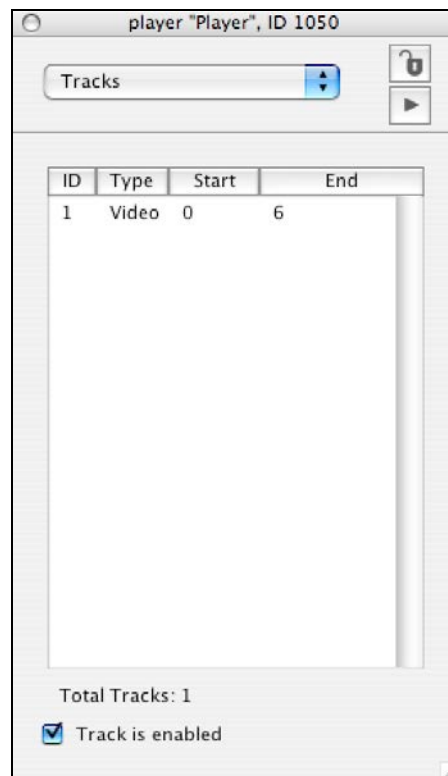


Fig. 28-2. Player Tracks.

TRACKS: The **TRACKS** section of the Quicktime player lists the various tracks on the player, indicating their ID numbers and type of track (audio or video). You can select any of the tracks and turn them on or off using the **Track is Enabled** (*enabledTracks*) checkbox at the bottom of the Property Inspector. Example script:

set the enabledTracks of player "My movie" to 1 -- enables only track 1 of the player

If you want to use a script to enable more than one track of a player, these need to be referred to on different lines. Thus:

set the enabledTracks of player "My movie" to 2 & return & 3

PLAYER CALLBACKS (Fig. 28-3)

Player Callbacks enables you to effect an action at particular times while the movie is running. In the **Time Index** column place the *currentTime* at which you want the action to occur. In the **Message** column put the command that you would like to take effect at that time.

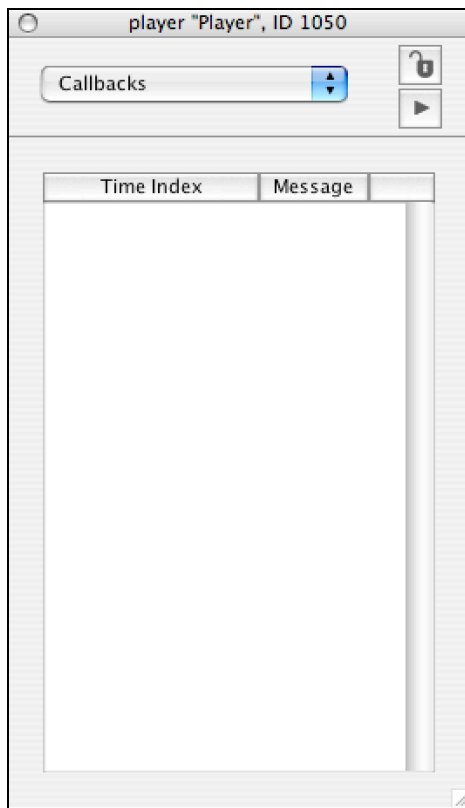


Fig. 28-3. Player Callbacks.



Fig. 28-4. Player QuickTime VR.

PLAYER QUICKTIME VR (Fig. 28-4)

Pan (*pan*) / **Tilt** (*tilt*) / and **Zoom** (*zoom*): By typing numbers into these 3 fields you are not permanently setting the pan (horizontal sweep), tilt (vertical sweep), and zoom of the Quicktime VR. Rather, you are just making a notation of what you would like the pan, tilt, and zoom to be, for future reference in a script. I.e., you would play around moving the Quicktime VR in Run Mode, and when you get a pan, tilt, and zoom that looks nice, determine these numbers via the Message Box:

put the pan of player 1
or
put the tilt of player 1
or
put the zoom of player 1

You would record these values in the pan, tilt, and zoom fields of the Property Inspector for future reference. If you now go to another card and return, the panorama reverts to its default values. So you need a script on opening the card to set the pan, tilt, and zoom to the desired values that you had notated. e.g.:

on openCard
set the pan of player "My panorama" to 140
set the tilt of player "My panorama" to 10
set the zoom of player "My panorama" to 45
end openCard

Your Quicktime VR may include more than just the panoramic movie. It may contain hotspots ingrained in the movie to cause some action when the hotspot is clicked on. For example, a door within a room in a Quicktime VR may contain a hotspot which when clicked on will effect an action, e.g., opening the door to a different scene. Hotspots are created in third party software that constructs Quicktime VR movies.

To get a list of the hotspots in the QTVR, write in the Message Box:

put the hotSpots of player "MyPanorama"

The list of hotspots, if there are any, will appear in the Message Box.

To determine which hotspots the mouse is over, write in the script editor of the player:

on hotspotClicked someSpot
put someSpot
end hotspotClicked

This will tell you the ID of the hotspot that you just clicked, e.g. it may be 70 or 60, etc.

The only thing now is to create a script telling LiveCode what to do when the hotspot is clicked:

on hotspotClicked someSpot
if someSpot is 70 then go to card "NewScene 1"
if someSpot is 60 then go to card "NewScene 2"

end hotspotClicked

PLAYER SIZE & POSITION

The **Lock Size and Position** checkbox locks the position and size of the player, so that it does not move around or resize inadvertently during development. When the box is checked, the player cannot be moved with the mouse. However, it can still be moved with the arrow keys when selected in Edit mode.

Locking fixes not only the position of the movie, but keeps its size the same when moving back and forth between cards.

AUDIOCLIPS PROPERTY INSPECTOR (Fig. 28-5)

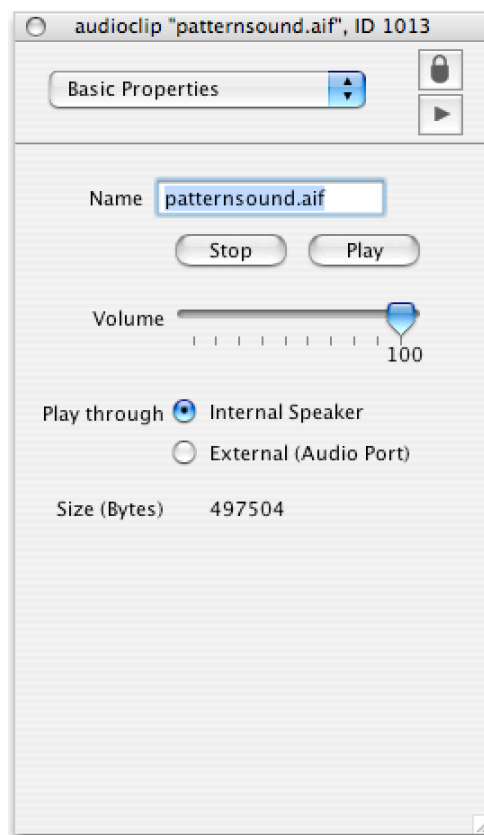


Fig. 28-5. AudioClips Property Inspector.

While audio files can be referenced through the Quicktime player, they can be imported (**File/Import As Control/Audio File**) where they become ingrained within the stack in the **audioClips** sections of the Application Browser until called upon by script. AudioClips are not objects. An audioClip's Property Inspector is accessed by right-clicking on the particular audioclip in the Application Browser. The audioclip Property Inspector has several Basic Properties:

NAME (*name*)

SIZE (in bytes) (*size*): Describes the size, in bytes, of the audioclip.

VOLUME (*playLoudness*): Enables setting the audioClip loudness from 0 to 100.

Sounds and Movies Scripting

Sounds and movies can both be played as referenced files through the **Quicktime Player** object. They can also be ingrained within LiveCode as **audioClips** and **videoClips**, as seen in the Application Browser.

AUDIOCLIPS

play audioClip
play stop
the playLoudness

play audioClip "MyMusic.aif" -- plays the audioClip
play stop -- stops the audioClip from playing

set the playLoudness to 100 -- without specifying an audioClip, this in general sets the speaker system loudness to 100 for all sounds. The *playLoudness* can be between 0 and 100.

set the playLoudness of audioClip "MyMusic.aif" to 60 -- When specifying a specific audioClip, this sets the loudness of just that specific audioClip.

VIDEOCLIPS

The QuickTime Player has more features than the videoClip and is less buggy. I suggest using the QuickTime Player for movies rather than the videoClip, which will not be discussed further.

QUICKTIME PLAYER

play/stop playing player
on playStarted
on playStopped
the playRate
the playLoudness

Examples:

play player 1 -- equivalent of *start player 1* and *set the paused of player 1 to false*. When a player is paused, *play player* resumes play from the paused spot.

stop playing player 1 -- equivalent of *set the paused of player 1 to true*

Scripts inside the player:

on playStarted -- takes effect when the movie starts.
 <do something>
end playStarted

on playStopped -- takes effect when the movie ends.
 <do something else>
end playStopped

One line of code can set the player playing or not playing:

set the paused of player "myPlayer" to not the paused of player "myPlayer"

You can also toggle pausing and resuming with the spacebar if **focus with keyboard** in the player Property Inspector is checked (*traversalOn* is *true*).

set the playRate of player 1 to 2 -- double speed
set the playLoudness of player 1 to 100 -- the maximum player loudness

the QTversion

The QTVersion returns the version of Quicktime on your computer.

if the QTVersion is "0.0" then answer "Quicktime is not installed."

CHAPTER 29. ABSOLUTE VS. REFERENCED FILE PATHS

The Image and Quicktime Player objects can respectively reference images and movies from outside LiveCode. The QuickTime Player, as mentioned, can also reference sounds. Clicking on the source (*filename*) folder in the **Basic Properties** section of the Image or Player Property Inspector asks you to pick a file; the Inspector then inscribes within the Property Inspector source folder the **long absolute path** to the image or movie, e.g.:

Hard Drive/Application folder/Images/MyPhoto.png – for the Image object
 Hard Drive/Application folder/Images/MyMovie.mov – for the QT Player object

Once the standalone is created, other computers may not recognize the long

absolute path, so it is necessary to change the long absolute path to the shorter **relative path** that starts within the folder in which your application resides. One way to do this is to create a folder for all the images and movies, titled, for instance, “IMAGES”. That folder should always be together with the stack file during development. That is, the stack file should reside in a folder together with another folder titled “IMAGES”, which contains the referenced movies:

FOLDER

```
MyMainstack.rev
IMAGES
    MyPhoto.png
    MyMovie.mov
```

Once this is done, you can manually shorten the paths within the Image and QT Player object **Source** field to a referenced path that begins within the folder in which the stack resides. The shorter referenced paths (note the absence of a forward slash at the beginning) would be:

Images/MyPhoto.png -- for the Image Object
Images/MyMovie.mov -- for the QT Player Object

Then, if the final standalone application is located together with the “IMAGES” folder, the pictures and movies should show up in the standalone on all computers.

Doing this by script can be done as follows:

set the filename of player 1 to “Images/MyMovie.mov”

If you want more than just an “Images” folder, but subfolders as well, this will work too:

FOLDER

```
MyMainstack.rev
Media
    Images
        MyPhoto.png
    Movies
        MyMovie.mov
```

The relative path would then read:

Media/Images/Movies/MyMovie.MyPhoto.png

At the time of this writing, LiveCode versions 6.7 and higher do not allow the creation of relative paths in the movie player on Macintosh. This presents certain

problems in displaying movies consistently in the stack and standalones created on Mac. There are several relatively easy ways to get around this at this time:

1. Work with a LiveCode version less than 6.7 on Mac until the problem is corrected. This will not only allow you to create the standalones as above, but also use Quicktime VR movies, which LC 6.7 and above will not show.
2. Work with the following script modification (on Mac) in a preopenCard script:

```
on preopenCard
  if the environment is "development" then
    set the filename of player 1 to "Images/MyMovie.mov"
  end if
  if the environment is "standalone application" then
    set the filename of player 1 to the defaultfolder & "/Images/MyMovie.mov"
  end if
end preopenCard
```

This will insure that the movie shows up in the stack ("development" environment) as well as in the "standalone application".

Work to address this issue of relative vs absolute paths is currently under way by the LiveCode team.

CHAPTER 30. THE GRAPHICS PROPERTY INSPECTOR

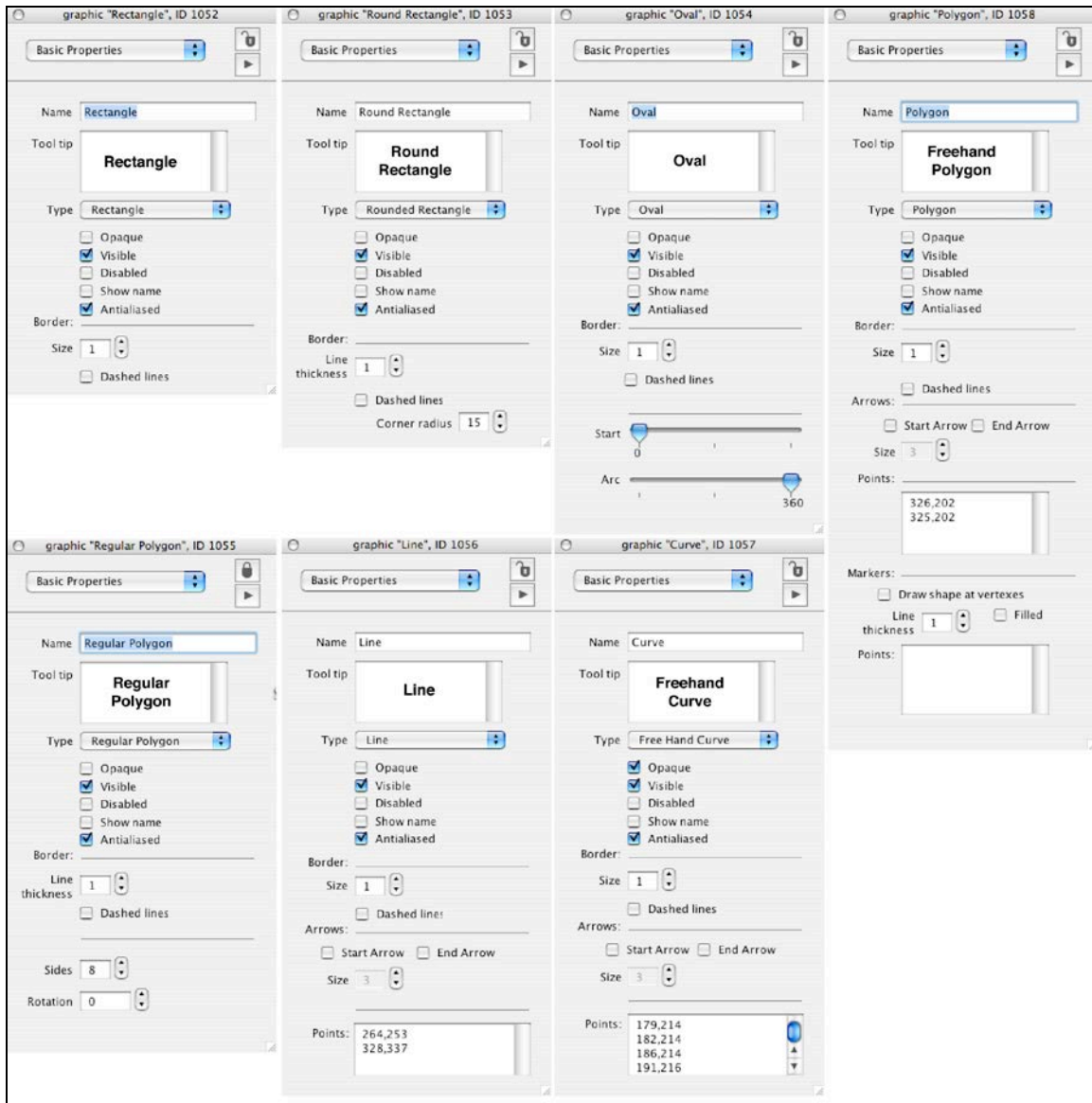


Fig. 30.1. Graphic Tools.

More sophisticated programs such as Adobe Illustrator or Photoshop can be used to create graphics and images to import into LiveCode, so these drawing tools will not be discussed here. You may want to play around with these controls, though, for an intuitive idea of how they work, for simpler illustrations.

CHAPTER 31. THE SCROLLBAR/ SLIDER/ LITTLE ARROWS/ AND PROGRESS BAR PROPERTY INSPECTORS (Fig. 31-1)

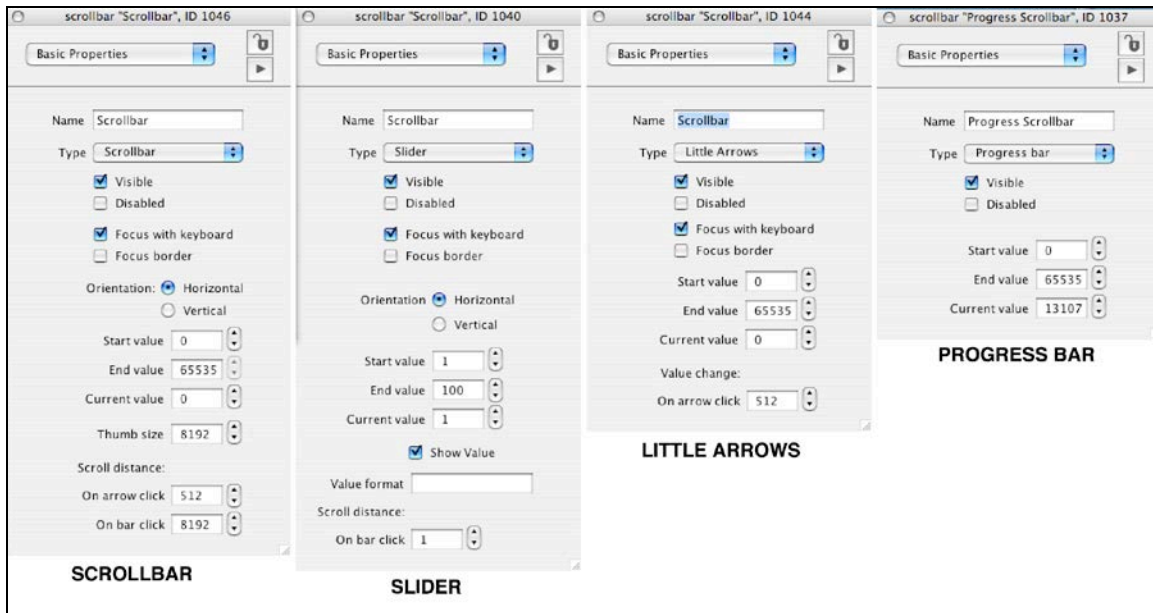


Fig. 31-1. Scroll Bar Property Inspectors.

PROGRESS BAR PROPERTY INSPECTOR

Progress bars are used to visually display the progress of a process over time.

START VALUE (*startValue*): Sets the number you would like to use as the starting value of the progress bar.

END VALUE (*endValue*): Sets the number you would like to use as the ending value of the progress bar.

CURRENT VALUE (*thumbPosition*): You may want to follow the progress of a timed operation, whose duration you know. For instance, the duration of a movie may be 18500, as indicated in the Quicktime Player's Property Inspector. Say you want to display the progress of the movie with your own Progress Bar. In the Progress Bar's **Basic Properties** you would indicate the **START** value (*startValue*) of the bar (the start time you would like the progress to begin at, which would be 0 in this case), as well as the **END VALUE** (*endValue*) of the operation, which would be 18500. The **CURRENT VALUE** (*thumbPosition*) is the position of the scrollbar at the present time. You would give a command that will start the movie and at the same time start the scroll bar moving. You need a script to do this. The following is a simple script that you could place in a button:

```
on mouseUp
  play player 1
  repeat until the currentTime of player 1 is 18500
    set the thumbPosition of scrollbar 1 to the currentTime of player 1
  end repeat
end mouseUp
```

SLIDER/SCROLLBAR PROPERTY INSPECTOR

START VALUE (*startValue*): Sets the starting number that you want on the slider.

END VALUE (*endValue*): Sets the ending number that you want on the slider.

CURRENT VALUE (*thumbPosition*): sets the current position of the knob of the slider

set the thumbPosition of scrollbar "Quantity" to 40

THUMB SIZE (*thumbSize*): set the length of the knob of the slider.

SHOW VALUE (*showValue*): Gives you the option of showing (*showValue* is *true*) or not showing (*showValue* is *false*) numbers on the slider. The default is *true* (checked box).

VALUE FORMAT (*numberFormat*): Lets you set the decimal places to which the number is shown on the slider. For instance: *#.00* is the dollar format, as if you were writing dollars and cents, with two digits after the decimal. A *numberFormat* of *0.0* shows only one digit after the decimal.

ON BAR CLICK (*pageInc*): For scrollbars and sliders, this sets how far the knob of the scrollbar or slider will go when clicking on the gray area of the bar.

As the Thumb of the Slide Bar is moved manually by the user, particular events can occur in the course of the Thumb movement, as directed by the Slide Bar script, such as the sound volume or values in a field. E.g.:

Sample script within a Slider "MyProgress" that has a **Start value** of 0 and an **End value** of 100.

```
on mouseStillDown
  put the thumbPosition of me into field "data"
end mouseStillDown
```

As the user slides the thumb, the values shown under the slide bar will also appear in the field "data".

Another way of scripting this is to use the word *scrollbarDrag*:

```
on scrollbarDrag myPosition
  put myPosition into field "data"
end scrollbarDrag
```


The **Little Arrows** control commonly is used to add or subtract numbers in a field. Drag a new field onto the card and name it “Counter”, and then drag a **Little Arrows** control and place it next to it. Insert the following script:

```
on scrollbarLineInc  
  subtract 1 from fld "data"  
end scrollbarLineInc
```

```
on scrollbarLineDec  
  add 1 to fld "data"  
end scrollbarLineDec
```

Clicking the little scrollbar will then increase or decrease the number in field “Counter”.

CHAPTER 32. MULTIPLE OBJECTS SELECTED PROPERTY INSPECTOR (Fig. 32-1)

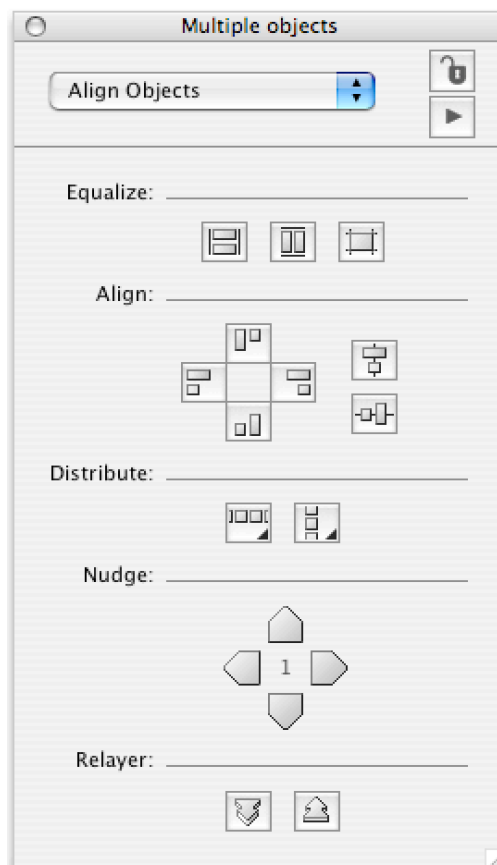


Fig. 32-1. Multiple Objects Property Inspector.

When you select more than one control on a card (not grouped) and open the Property Inspector, you will find that the Property Inspector is now named **“Multiple”** (**Multiple Objects, Multiple Buttons, Multiple Fields, Multiple Images**, etc. as the case may be). With this **Multiple** Property Inspector you can change certain properties of all the controls at once just as you can for individual objects. Most importantly, there is a section of the **Multiple** Property Inspectors that is termed **Align Objects** (**Fig. 32-1**), which enables you to align the selected objects as you see fit. For instance:

Place several buttons on a card and experiment with the Align features, which are intuitive.

CHAPTER 33. GROUPS PROPERTY INSPECTOR

GROUP BASIC PROPERTIES (Fig. 33-1)

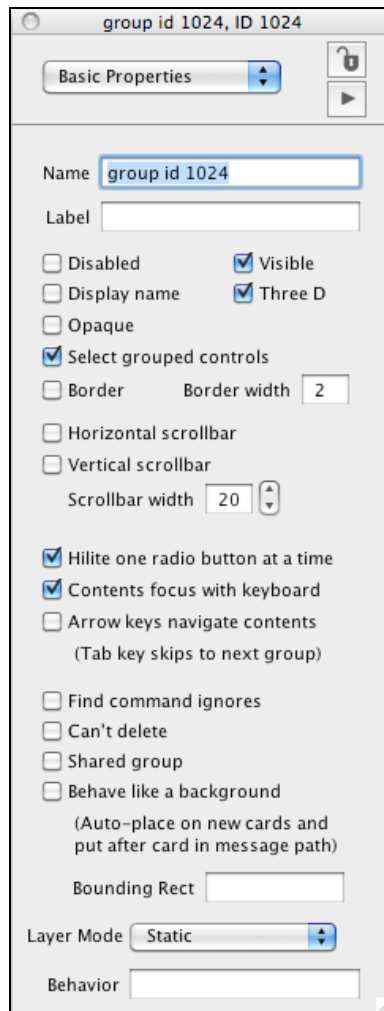


Fig. 33-1. Groups Property Inspector.

DISPLAY NAME (*showName*): Shows the name of the group in small letters to

the upper left of the group.

HORIZONTAL AND VERTICAL SCROLLBARS (*hScrollbar; vScrollbar*): Yes, you can put a scrollbar on a group as a whole and scroll the entire group with all its buttons and what ever else you want to include in the group. Scroll bars on text fields are quite familiar. But what if you want to create a group that contains buttons or other objects that you want to scroll as a whole, not just text? You can do this by aligning the objects vertically or horizontally, grouping them and adding a vertical or horizontal scrollbar to the group.

To prevent the dimensions of the group from automatically resizing when moving back and forth between cards, be sure to check off the **Lock Size and Position** box in the **Size and Position** section of the group Property Inspector.

HILITE ONE RADIO BUTTON AT A TIME (*radioBehavior*): Checking this feature insures that anytime one radio button in a group is hilited, the others are dehilited. It's best to leave this checked, for after all, that is what radio buttons are for. The default *radioBehavior* is *true*.

Note also that when a radio button is part of a group that is set as a background, you want to make clear whether or not clicking on a radio button on one card will simultaneously hilite the same button on all the other cards containing the group. Usually you want the radio buttons on each card to act independently of the other cards. In that case, keep the each radio button's **Shared Hilite** box unchecked (the default state) in the radio button's **Basic Properties** section.

FIND COMMAND IGNORES (*dontSearch*): When checked, this instructs LiveCode to skip any fields in this group when *find* commands are issued.

BEHAVE LIKE A BACKGROUND (*backgroundBehavior*): A very important feature!! If this is checked, then anytime you create a new card, the group will automatically be placed on the new card.

SECTION 4. THE MENU BARS

CHAPTER 34. THE LIVECODE MENU BAR

This chapter covers only key selected features of the LiveCode Menu Bar. Items that are obvious, already covered, well-known, of lesser interest, or are beyond the scope of this book are not included.

LIVECODE

LIVECODE/ PREFERENCES/GENERAL

Property labels are: Let's you choose whether you want the words for properties in the Property Inspectors to be descriptive (the default) or to be the actual script words used for the properties (as seen with the tooltip).

Arrow keys navigate through cards: If checked, this provides an easy way to navigate through the stack's cards, using the right and left arrow keys.

Image editor: Choose the image editor you want to use (e.g., Photoshop) to modify imported images

FILE

FILE/ NEW MAINSTACK: Creates a new mainstack.

FILE/ NEW SUBSTACK: Creates a new substack of the mainstack.

FILE/CLOSE: If there are more than one stack open, this closes the selected stack, as opposed to **Quit** (Macintosh) or **Exit** (Windows), which closes LiveCode and all the stacks that are open.

FILE/ IMPORT AS CONTROL (Fig. 34-1)

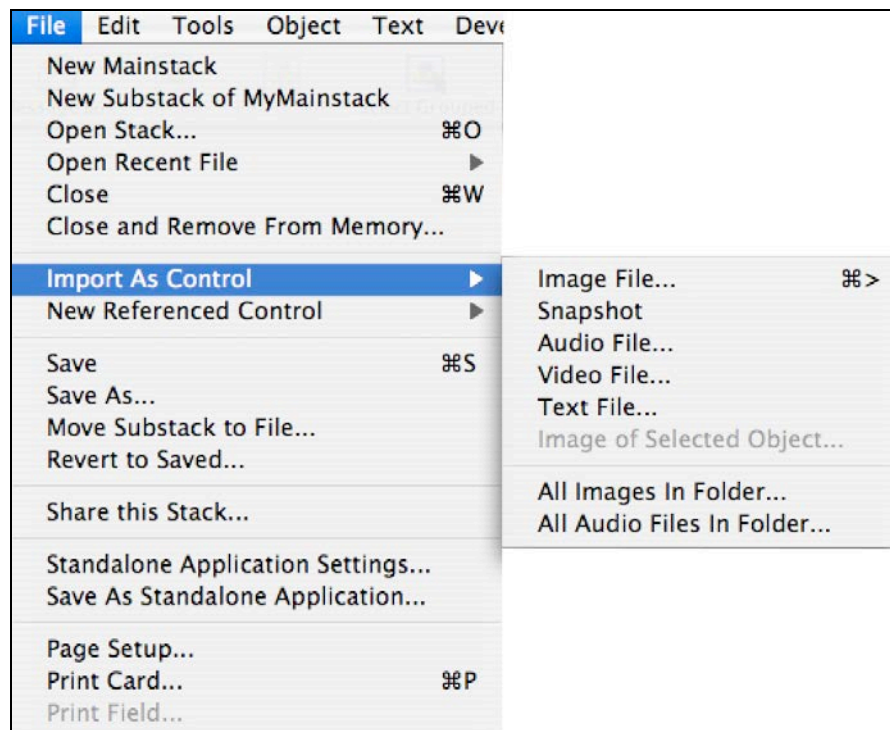


Fig. 34-1. File Import As Control.

Import as Control Imports the chosen files directly into LiveCode as part of the stack, rather than sitting outside the stack and having to be referred to. Directly imported files don't have to be stored separately outside the standalone application you create. The kinds of files you can directly import include:

Image file...: Can import JPEG, PNG, BMP, GIF (can be animated GIF), or PICT, among other less used formats. PICT, however, can only be used on Macintosh. LiveCode will ask you if you want to convert the PICT image to a PNG.

Snapshot...: Takes a snapshot of any area of the screen outside LiveCode and places it on the card! When you select **Snapshot**, LiveCode is hidden, so you are restricted to selecting the screenshot of areas outside LiveCode. However, you can still use a script to capture a screen shot of a card:

import snapshot from rect (the rect of this stack)

Right-clicking on the image will bring up the option of entering your image editor, where you can modify the image, e.g. making it a thumbnail image.

Audio file...: Can be AIF, WAV, or AU.

Video file...: Imports , as videoClips, Quicktime and AVI movies, as well as .mp4 movies, which take up less space than Quicktime (.mov) movies. It is more reliable, however to reference movies than to import them.

Image of selected object...: Places on the card an image of any selected object on the card.

All images in folder...: Takes all the images in a selected folder and places them on the card.

All audio files in folder...: Takes all the audio files in a selected folder and places them in the Application Browser under **AudioClips**.

FILE/ NEW REFERENCED CONTROL (Fig. 34-2)

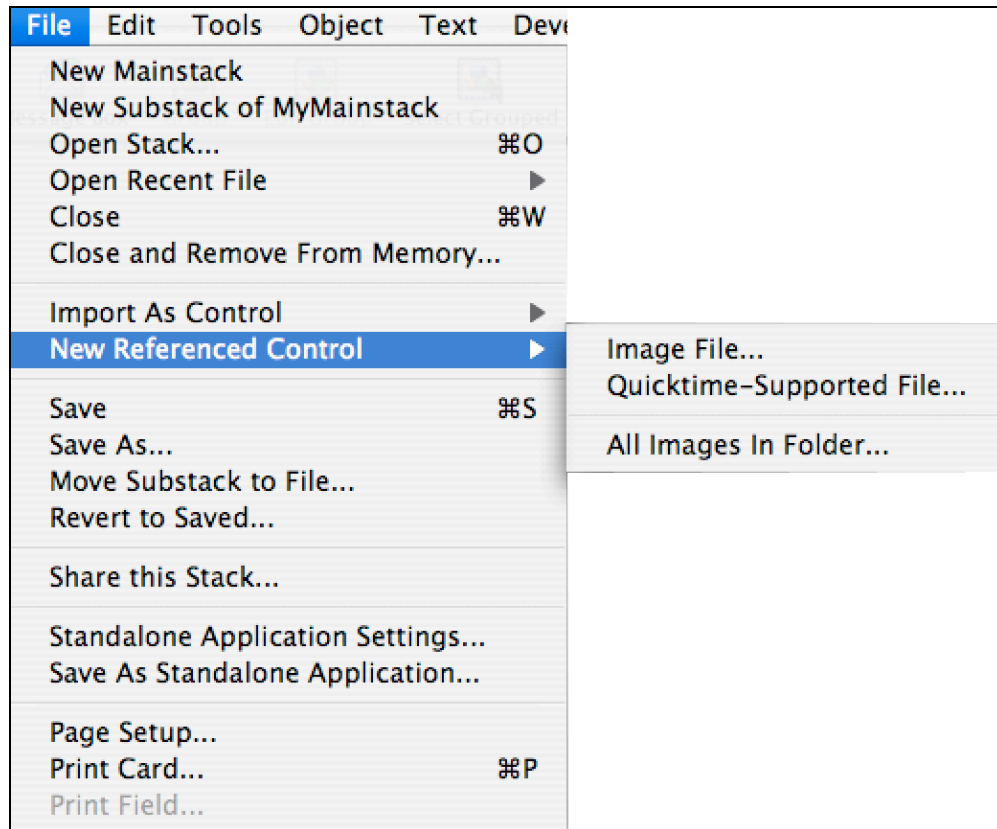


Fig. 34-2. New Referenced Control.

Image file: Brings a referenced image onto the card within an Image Area Object, along with its **Source** (the path to the stack).

Quicktime-supported file: Brings a referenced Quicktime (.mov) or AVI (.avi) movies onto the card within a Quicktime Player object, showing its **Source** (the path to the stack). You can also use a Quicktime Player to import sounds, particularly in .aif or .wav format.

All Images in Folder: Brings in all the images in a folder onto the card, each within its own Image Area Object, including each one's referenced **Source** (path to the stack).

FILE/ SAVE: Saves the stack, overwriting any previous version of the stack.

FILE/ SAVE AS: Saves the stack as a new stack, but preserves the previous file. Both SAVE and SAVE AS give you the option of saving the stack as an older version of LiveCode.

FILE/ MOVE SUBSTACK TO FILE: Changes a substack to a new mainstack, independent of the original mainstack

FILE/ REVERT TO SAVED: Reverts to the previous version of the stack if you haven't already saved the updated changes.

FILE/ STANDALONE APPLICATION SETTINGS

In building a standalone application, LiveCode automatically places its engine directly within the standalone, so you do not need a separate LiveCode player to run a standalone. **When a standalone application is created, it is important to note that you cannot later save changes to the standalone's mainstack, only to its substacks.** For that reason, many developers only use the mainstack as just one card, a simple splash screen that connects with the mainstack's substacks, where all the action resides and the saving of data can take place.

FILE/ STANDALONE APPL SETTINGS/ GENERAL SETTINGS (Fig. 34-3)

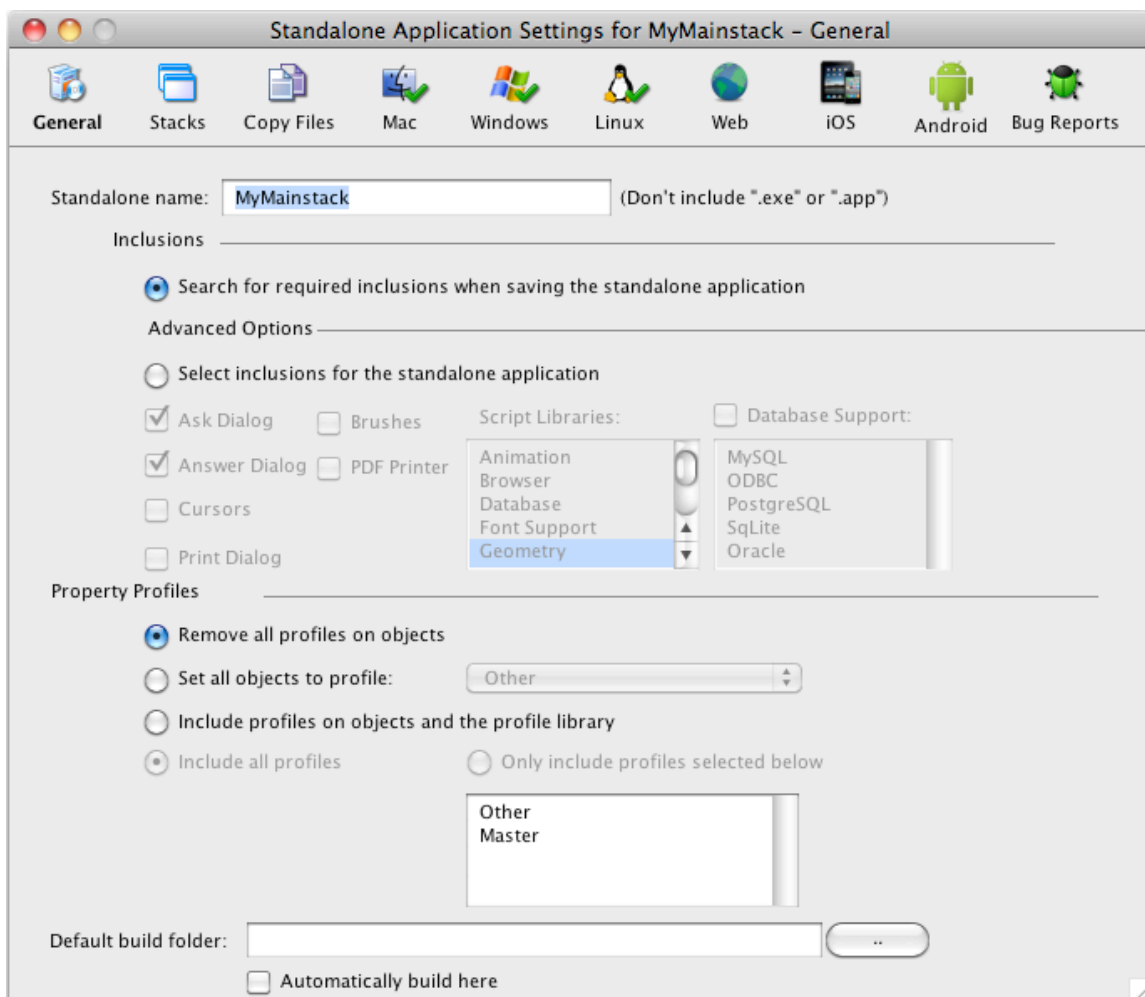


Fig. 34-3. Standalone Application Settings.

Standalone name: The name you give to the standalone in the standalone settings name box will be the name applied to the Mac and Windows standalone

applications, as well as to the folder that contains the Mac and Windows standalone folders. If you don't specify a standalone name, the mainstack's name will be used by default. (Fig. 34-3).

FILE/ STANDALONE APPL SETTINGS/ STACKS SETTINGS (Fig. 34-4)

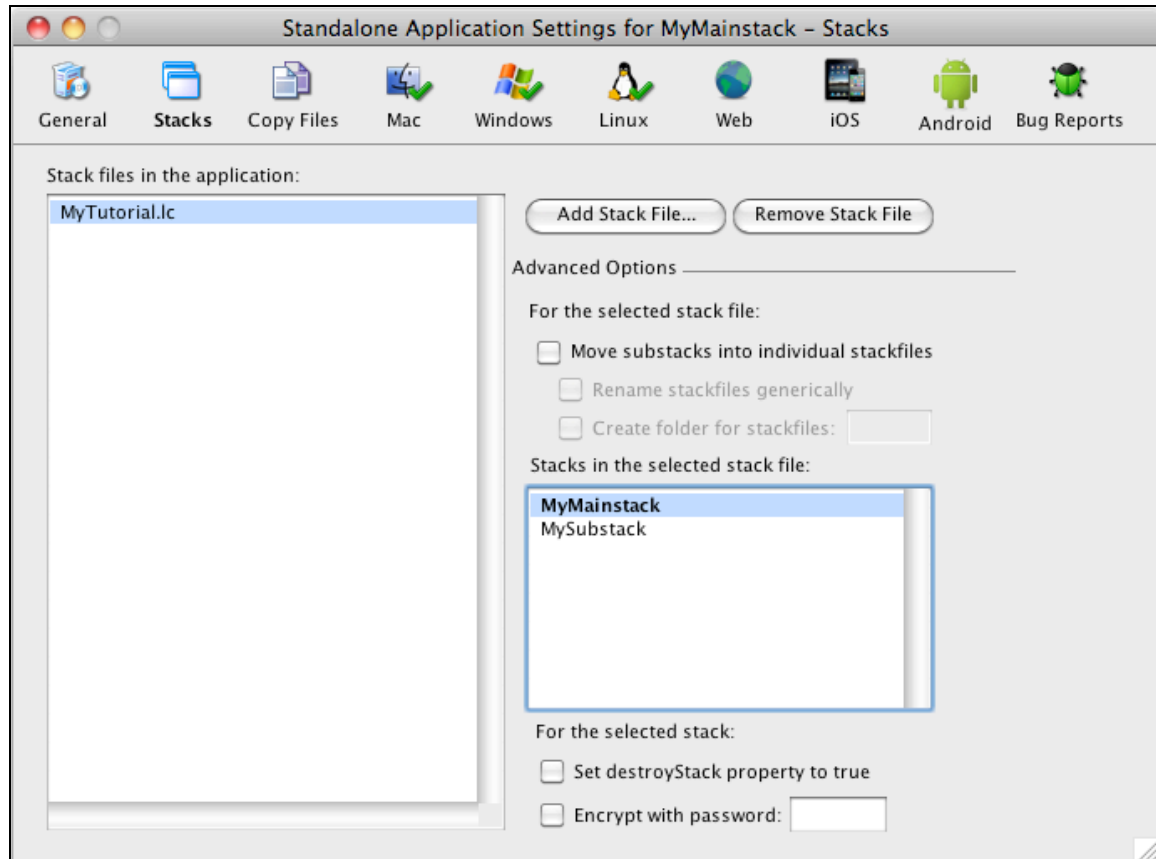


Fig. 34-4. Standalone Settings/Stacks.

Move substacks into individual stackfiles: Important!! Livecode normally stores the mainstack and all substacks in a single file. Check this box to insure that changes made by the user in the standalone can be saved. As previously mentioned, in a standalone, changes cannot be saved in the mainstack. To insure that changes can be saved, it is important that anything that intended for the user to change in the standalone (e.g. text in a text field) is placed in a substack, and the substacks are saved as individual files. Use the mainstack as a simple single card which does not change and which connects with the substacks. Checking this box insures that changes can be made by the user in the standalone.

The icon for a Macintosh standalone is the only file you see; i.e., the substacks are hidden. However, Windows shows not only the application icon but also each substack. To avoid confusion with which icon to click on to open the program, you may want also check off "**Rename stackfiles generically**", which

automatically renames the substacks with a boring name, so as not to entice the user to click on them. (This is not necessary for Mac, since Mac substacks are hidden from view in the standalone.) You can further isolate the substacks from the main application icon by checking off “**Create folder for stackfiles**”, which places the substacks in a separate folder, where they are not immediately next to the main application icon.

One more thing one has to be done to insure that changes are saved in a standalone. There has to be a script in the substack indicating that the substack should be saved prior to quitting. For instance, the substack could have as part of its stack script:

```
on closeStack
  save this stack
end closeStack
```

Encrypt with password: After creating a standalone, the Windows version’s stacks are visible in the application’s folder; on the Macintosh they are located within the application package. To prevent Windows users who have LiveCode from viewing the script in these stacks, pick the substack(s) you want to protect and assign a password to. The user will not be able to view the scripts in that stack without the password.

FILE/ STANDALONE APPL SETTINGS/ MAC (Fig. 34-5)

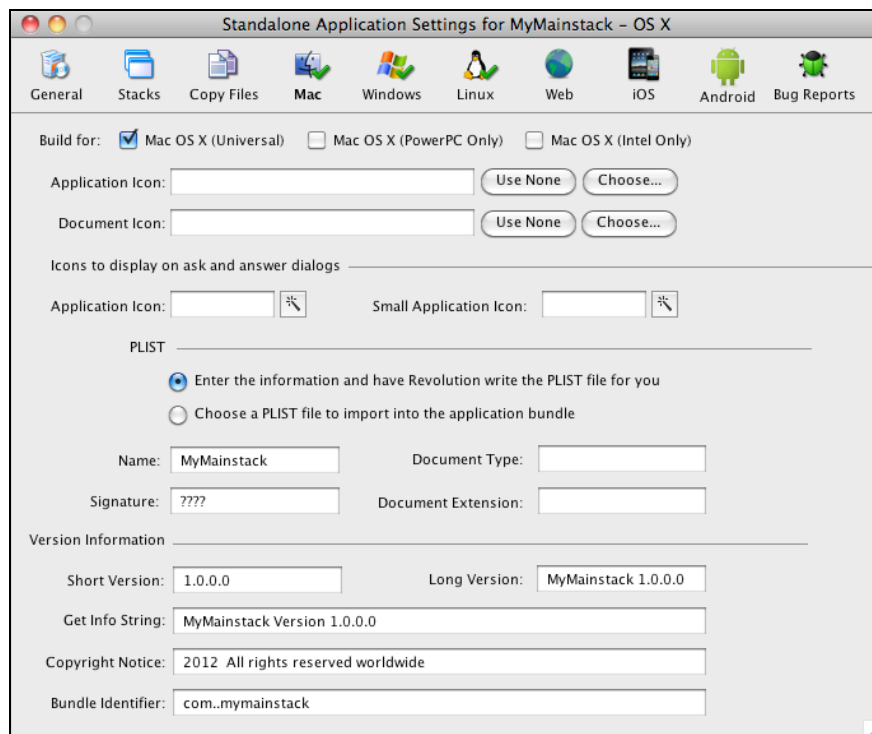


Fig. 34-5. Standalone Settings/Mac.

Build for: Check **MacOS X (Intel)** if you are building for Mac.

Application icon: Choose an icon for the Mac OS X application (use .icns icon format for Mac (.ico format for Windows). An easy and immediate way to apply an icon to your Mac application is simply to slide the image file of interest onto the generic image in the upper left corner of the built application's **Get Information** window.

Icons to display on ask and answer dialogs:

Application icon: Use .icns icon format for Mac (.ico format for Windows)

When you use ask or answer dialogs in a stack, an icon appears at the left of the dialog window. The default is a LiveCode icon. In a standalone, though, the icon is missing unless you supply one here, in which ask and answer dialogs will display your custom standalone icon in these dialogs if you wish. These don't need to be regular icon files; just an image of the correct size stored somewhere in the stack. **Regular** size is 64x64 and **small** is 32x32. There is more info in the dictionary under "gRevSmallApplIcon" and "gRevApplIcon".

FILE/ STANDALONE APPL SETTINGS/ WINDOWS SETTINGS (Fig. 34-6)

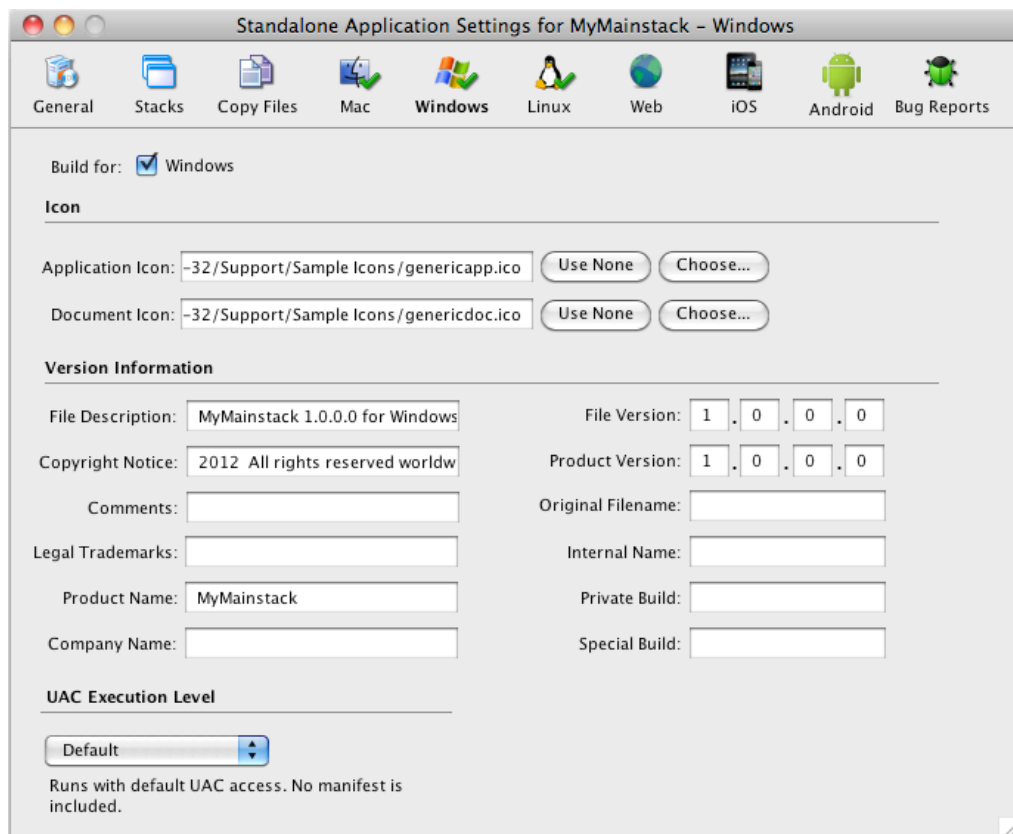


Fig. 34-6. Standalone Settings/Windows.

Build for: Check if you want a build for Windows.

Application icon: Use .ico icon format for Windows (.icns format for Mac)

FILE/ SAVE AS STANDALONE APPLICATION

Asks you where to save the standalone. You must first indicate, under **Standalone Application Settings**, whether you wish to save for Mac OS, OSX, Windows, Linux, Web, iOS and/or Android.

EDIT

EDIT/ SELECT GROUPED CONTROLS: When this is **checked**, clicking on one of the controls in a group does not select the group as a whole but the individual control, enabling you to make alterations to the size and position of the particular control. You can also remove various controls from the group but cannot add a new control. When **Select Grouped Controls** is **unchecked**, the whole group is selected, allowing resizing and positioning of the group as a whole, as well as access to the group's Property Inspector and script.

If you want more flexibility in editing, including the ability to add controls to a Group, select **Edit Group** from the **Object** menu, or click on **Edit Group** in the Icon Tools bar. The only disadvantage of **Edit Group** in editing is that all objects on the card other than the selected group are rendered invisible during the editing process.

EDIT/ FIND AND REPLACE... (Fig. 34-7)

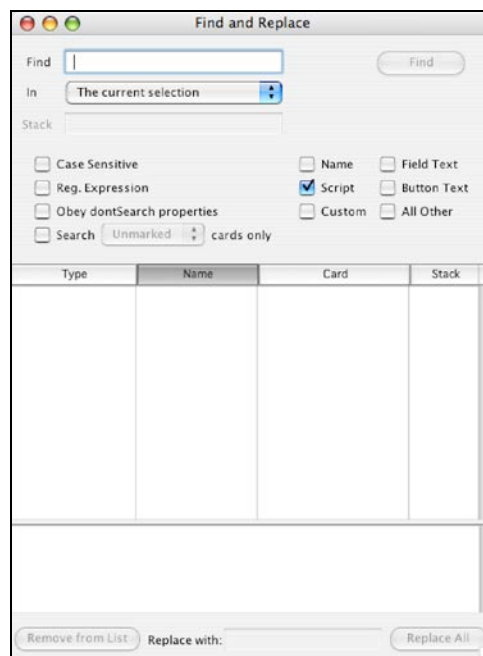


Fig. 34-7. Find and Replace.

Find: Prompts you to type in this field the word(s) you want to find.

In: Select from the pulldown menu the areas where you want to search – **the current selection, this card, this stack, etc.**

Case Sensitive: (if checked, the search is exact in regard to case – e.g. “d’ versus “D”)

Obey Don’t Search Properties: Gives directions not to search the text in fields that have **Find Command Ignores** in their Property Inspectors.

Search Marked/ Unmarked Cards: Limits the search to either marked or unmarked cards.

Name, Script, Custom, Field, Text, Button Text, All Other: Check each of these areas in which you want to search. **Custom** means that the search should include **Custom Properties** as well.

Type, Name, Card, Stack Columns: Once the search is done (by clicking on the “Find” button), LiveCode indicates the places where the search term was found. Clicking on the found word(s) will take you to the spot where the word(s) appear(s). The number of finds is displayed right under the **Find** button.

Remove From List Button: Removes a selected line from the list of locations in the above columns.

Replace With: Type in this field what word(s) you would like to use as a replacement for the search word(s).

Replace All: Clicking this button replaces all the found words with the words in the **Replace With** field. Sometimes, when the number of occurrences of the search term is large, you have go through the search and replace process several times, until clicking on the **Find** button results in no further finds of the search word(s).

TOOLS

TOOLS/ TOOLS PALETTE: Shows or hides the Tools Palette.

TOOLS/ APPLICATION BROWSER (Fig. 2-5): Shows the Application Browser. See **Chapter 5** for a description of the Application Browser.

TOOLS/ MENU BUILDER: Not discussed in this book. Personally, I have found it easier to construct my own menus outside the Menu Builder.

TOOLS/ MESSAGE BOX: Toggles the showing and hiding of the Message Box.

OBJECT

OBJECT/ GROUP, UNGROUP SELECTED: Places the selected objects into a group, or ungroups them.

OBJECT/ EDIT GROUP: Hides all objects on the card except the selected group. You can then focus on editing the group, whether changing the properties of objects within the group or adding or removing controls from the group.

OBJECT/ REMOVE GROUP: When a selected group is a background on a number of cards, selecting **Remove Group** will remove that group from the card of interest, without removing the group from the other cards. It is important to select **Remove Group** rather than just pressing the keyboard Delete button or choosing **Clear Objects** from the LiveCode Edit menu. Those choices would delete the group from all cards in the stack, and LiveCode will issue a warning that this will occur if you proceed.

OBJECT/ PLACE GROUP: When you have a group that you want to place on a card that does not yet have that group on it, selecting **Place Group** puts the group on that card. Thus, **Remove Group** and **Place Group** provide a way of selectively removing or placing a group on one card, without affecting the group on other cards. It is helpful to name each group; otherwise it may be difficult to distinguish one group ID from another.

OBJECT/ FLIP: Flips an image horizontally or vertically, whether or not the image is imported as a control or referenced.

OBJECT/ ROTATE: Rotates an image any degree you want. It works only on images imported as a control, not on referenced images.

OBJECT/ RESHAPE GRAPHIC: Places handles on a drawn freehand or freehand polygon vector graphic, enabling it to be reshaped.

OBJECT/ SEND TO BACK: Moves a selected object back to the bottom of the stacking order right against the card itself.

OBJECT/ MOVE BACKWARD: Moves the selected object back one layer.

OBJECT/ MOVE FORWARD: Moves the selected object forward one layer.

OBJECT/ BRING TO FRONT: Moves the selected object to the front layer of all the objects on the card (farthest from the card).

TEXT

TEXT/ PLAIN, BOLD, ITALIC, UNDERLINE, STRIKEOUT, BOX, 3D BOX: Changes the style of the selected button or field's text, including individual words in the field.

TEXT/ LINK: Groups words as a link, which can be activated as a whole. E.g., if the linked words are **computer sales**, the *clickText* on clicking on either of these words would be **computer sales** rather than the individual word **computer** or **sales**.

TEXT/ SUBSCRIPT, SUPERScript: Changes selected characters in a field to subscripts or superscripts.

TEXT/ FONT, SIZE, COLOR, ALIGN: Changes the font, size, color, or alignment of selected text in a field or in a button name, overriding any settings made within the stack, card, or field Property Inspector. For instance, if the field has a particular text setting and you want to change the font, size, style, or color of certain words in that field without affecting the other words in the field, you would do that through the **Text** menu, which can change the properties of individual words in the field, while the remainder of the field's words follow the settings of the field's Property Inspector.

DEVELOPMENT

DEVELOPMENT/ IMAGE LIBRARY (Fig. 34-8): Has libraries of images that you can place on a card by selecting **Place Image**. If you choose **Place Reference**, it will place it as an iconed button, rather than an image. If you have an image that you imported into LiveCode, you can add it to your personal library by choosing **Import Selected**. You can import images from outside LiveCode into your personal library by choosing **Import File**. **Import File** can import an entire set of pictures in a folder if you click the first and last picture in the folder with the Shift key down.



Fig. 34-8. Images Library.

DEVELOPMENT/ PLUGINS: Accesses custom tool stacks in the Plugins folder. To make your own plugin, create a stack that you would like to use as a plugin, and place it in the LiveCode Plugins folder. Then, when you open LiveCode, the stack (the name you gave it in the stack's Property Inspector) will open by selecting it from the **Development/ Plugins** pulldown menu. If you want your plugin to open automatically just by opening LiveCode, select **Development/ Plugins/ Plugin Settings** (Fig. 34-9) from the LiveCode Menu bar, select your plugin, and choose **LiveCode Starts Up**. By default, the plugin stack opens as a palette. You have options within the **Plugin Settings** dialog box to open it as a palette, modeless dialog box, modal dialog box, or invisible.

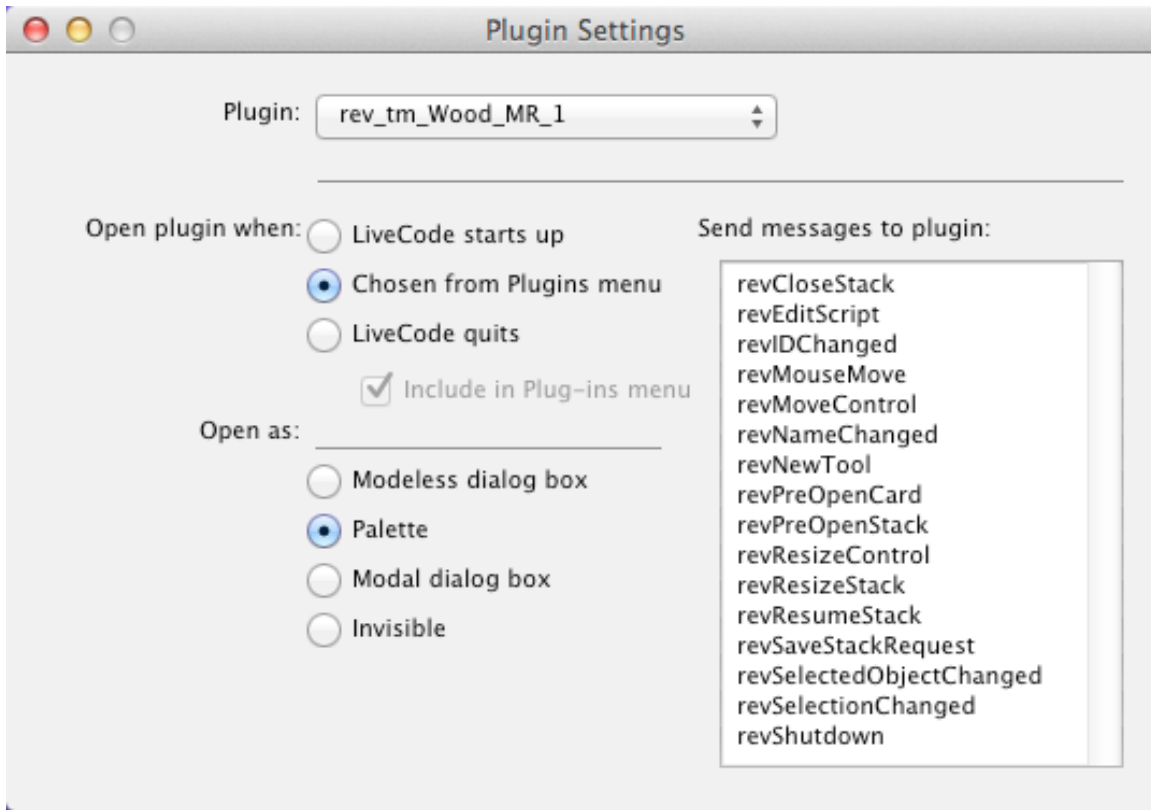


Fig. 34.9. Plugin Settings.

Apart from **opening** the plugin stack, you need to do one more thing. To actually **use** plugin stack “MyPluginStack” you need to issue the command:

start using stack “MyPluginStack”

The *start using* command is necessary any time you want to access another stack that is not part of the stack file that you are in.

If you build a standalone and want your plugin stack to be included (for example, to access scripts in the plugin stack), then include the plugin stack together with the standalone. As long as the standalone application has the *start using stack “MyPluginStack”* within it, the plugin stack will be accessed.

VIEW

VIEW/ GO FIRST (Command-1): Goes to the first card in the stack.

VIEW/ GO PREV (Command-2): Goes to the previous card in the stacking order (e.g., from card 5 to card 4).

VIEW/ GO NEXT (Command-3): Goes to the next card in the stacking

order (e.g., from card 4 to card 5).

VIEW/ GO LAST (Command-4): Goes to the last card in the stack.

VIEW/ GO RECENT: Goes to the card you just came from. The equivalent of *go back*.

VIEW/ TOOLBAR TEXT: Shows the text accompanying the Icon Toolbar.

VIEW/ TOOLBAR ICONS: Shows the icons of the Icon Toolbar, which lies just under the LiveCode menubar.

VIEW/ PALETTES: Shows or hides the various palettes that are open.

VIEW/ RULERS: Shows or hides the horizontal and vertical rulers at the edges of the stack.

VIEW/ GRID: The grid is not actually visible, but it's "snap-to" effect can be turned on or off, as seen by dragging an object across the card. (You may have to increase the grid spacing in the **PREFERENCES/APPEARANCE** pane to see the effect clearly.)

VIEW/ BACKDROP: Activating the backdrop fills the screen outside the stack with a backdrop color or pattern. This can help remove from view distracting desktop items. Color and pattern details of the Backdrop are set in the LiveCode's **Preferences** section, under **Appearance**. Be sure

VIEW/BACKDROP is unchecked if you don't want to see a backdrop when you create a new stack.

VIEW/ LIVECODE UI ELEMENTS IN LISTS: Allows you to see, in the Application Browser, all the LiveCode development environment stacks that are open and play a role behind the scenes in controlling LiveCode. Be sure this is unchecked unless you want to go nuts looking what's in the Application Browser.

VIEW/ SHOW INVISIBLE OBJECTS: When you have hidden objects in your stack and want to locate them, **Show Invisible Objects** makes all of them visible. Be sure to turn this off when you don't need it. Otherwise, you will wonder why you can't make an object invisible.

HELP

Gives you access to the LiveCode **Dictionary** and other resources.

HELP/ BUY A LICENSE: Connects to the LiveCode Internet store.

THE SCRIPT MENU BAR

SCRIPT EDIT/ QUICKFIND: Opens a **Find** field at the bottom of the Script Editor to assist in finding words in the Script. You can also open this **Find** field with Command/F (Macintosh) or Control/F (Windows).

SCRIPT EDIT/ FIND AND REPLACE: Finds and replaces specific words in the script.

SCRIPT EDIT/ FIND SELECTION: Finds selected words in other areas of the script.

THE ICON TOOL BAR

Inspector: Opens the Property Inspector of any selected object.

Code: Opens the script of any selected object.

Message Box: Toggles the Message Box open or closed.

Group: Toggles grouping or ungrouping of objects.

Edit Group: Toggles a group into or out of editing mode.

Select Group: Toggles the selection of the group as a whole vs the ability to select the individual controls within the group.

User Samples: Connects to a list of example stacks created by LiveCode users (Fig. 34-10).



Fig. 34-10. Development Revonline Browser.

REFERENCES

GENERAL

The LiveCode User Guide. (Accessible from the LiveCode Help Menu or printed version).

Schonewill, M. *Programming LiveCode for the Real Beginner*, 2013, Economy-x-Talk. *AnimationEngine*, Derbrill. (animation techniques for LiveCode.)

<http://www.hyperactivesw.com/revscriptconf/scriptingconferences.html>. (Hyperactive Software scripting conferences.)

<http://livecode.byu.edu/indexgeneric.php>. (Brigham Young University online course.)

<http://forums.livecode.com/index.php>. (LiveCode Forums.)

<http://lessons.runrev.com>. (LiveCode lessons.)

<http://livecode.com/resources/>. (LiveCode Resources.)

<https://www.youtube.com/user/RunRevLtd/videos>. (YouTube videos.)

<http://newsletters.livecode.com/september/issue57/newsletter1.php>. (arrays)

<http://lessons.runrev.com/m/datagrid>. (Datagrid instruction.)

<http://quality.runrev.com/>, (Bugs)

MOBILE DEVICES

Holgate, C. *LiveCode Mobile Development: Beginner's Guide*, 2012, Packt Publishing.

Lavieri, E. *LiveCode Mobile Development: Hotshot*, 2013, Packt Publishing.
Lavieri, E. *LiveCode Mobile Development Cookbook*, 2014, Packt Publishing.
MobGUI. (Custom controls for mobile devices.)
<http://lessons.runrev.com/m/2571/l/23275-how-do-i-become-an-ios-developer>. How do I
Become an iOS Developer?