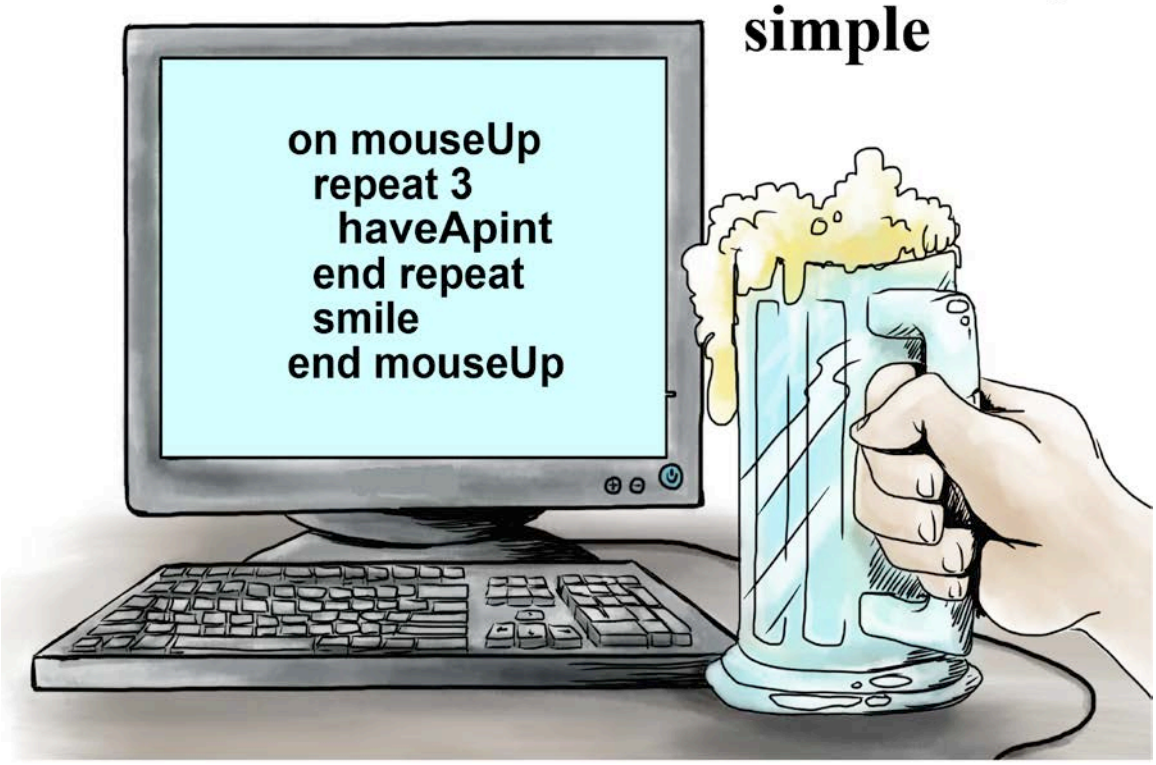
LiveCode Lite

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

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

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

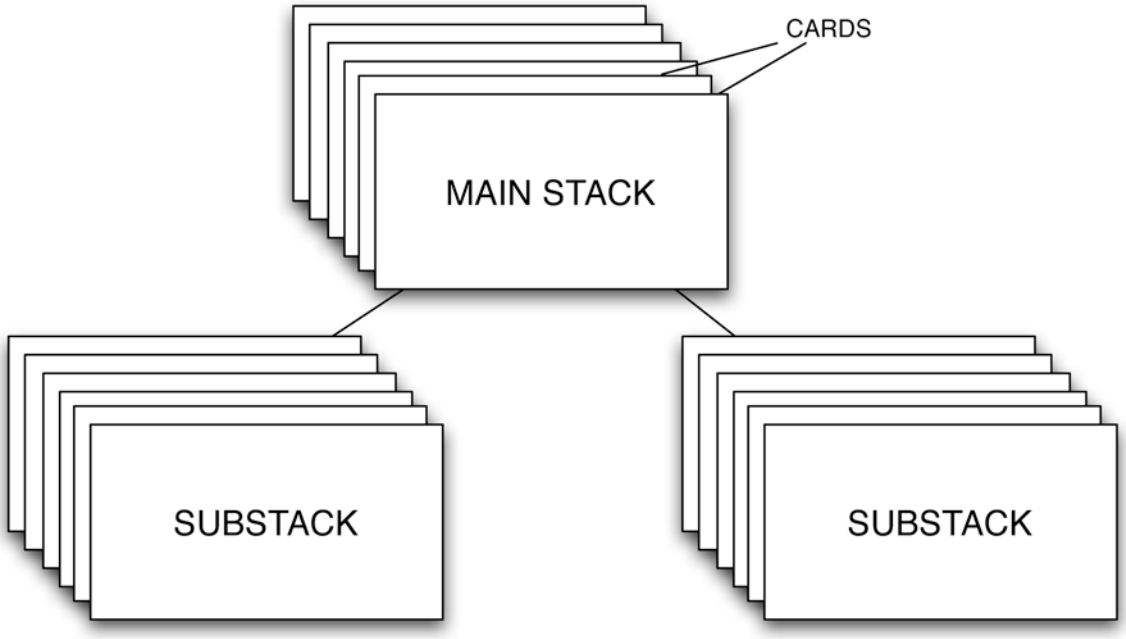


Fig. 2-1.

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 reshow this Help window, select **HELP/START CENTER**.

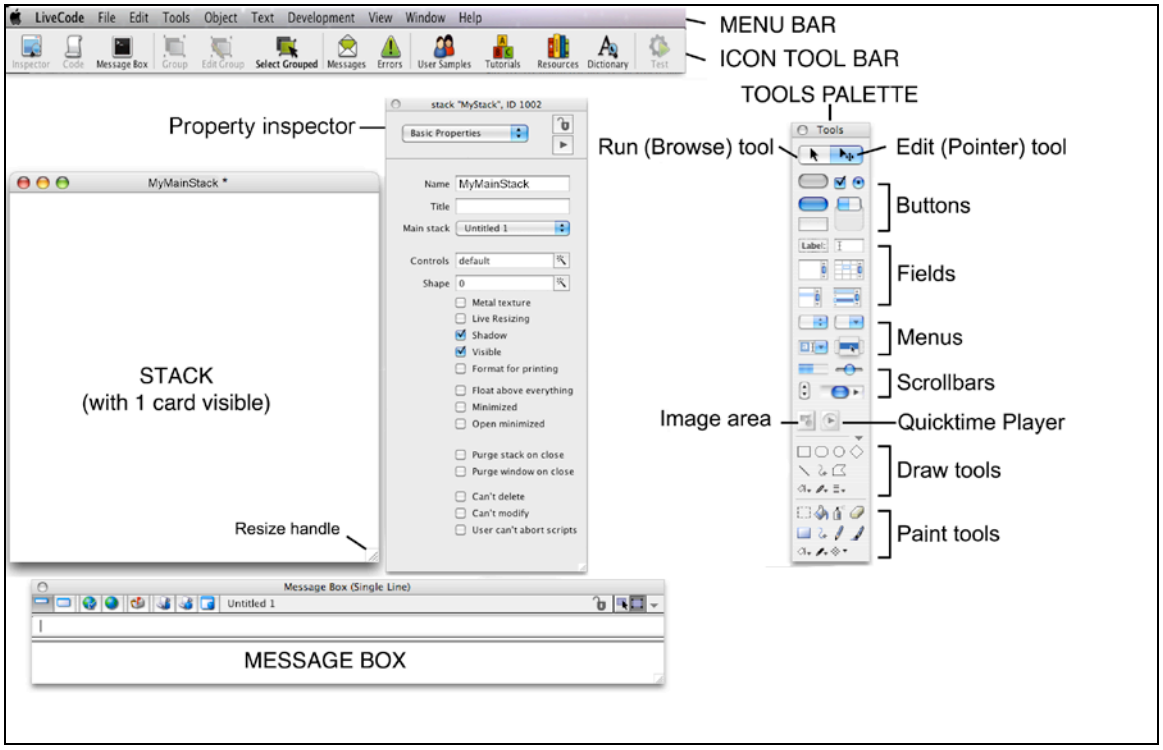


Fig. 2-2.

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**).

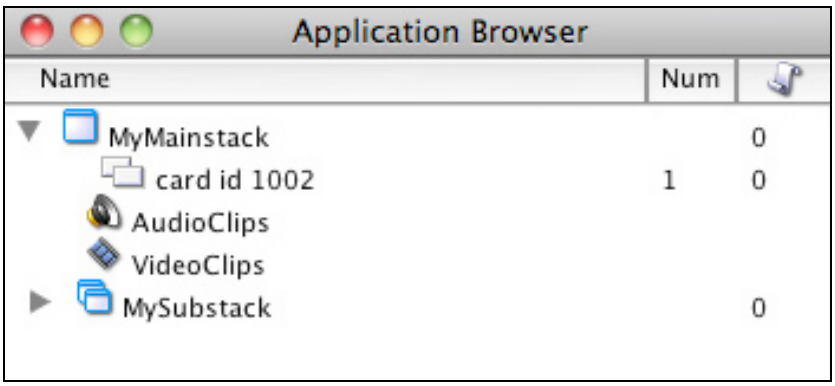


Fig. 2.3. Application Browser

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**.

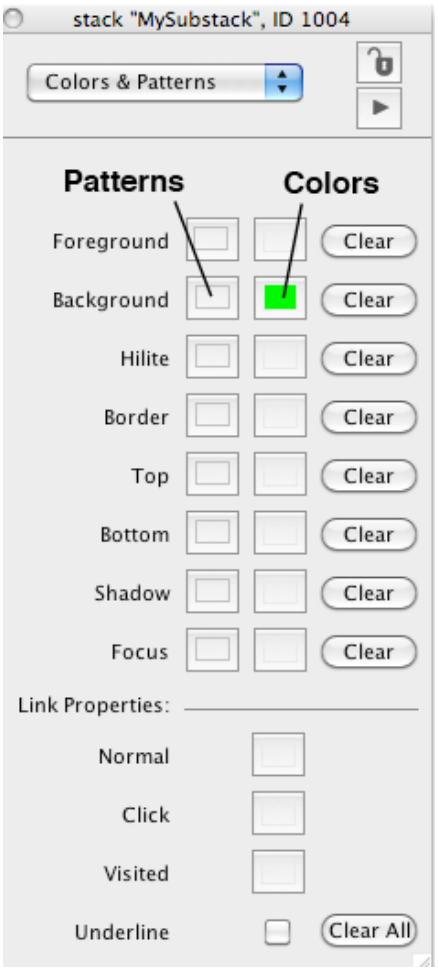


Fig. 2-4. Colors and Patterns

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.

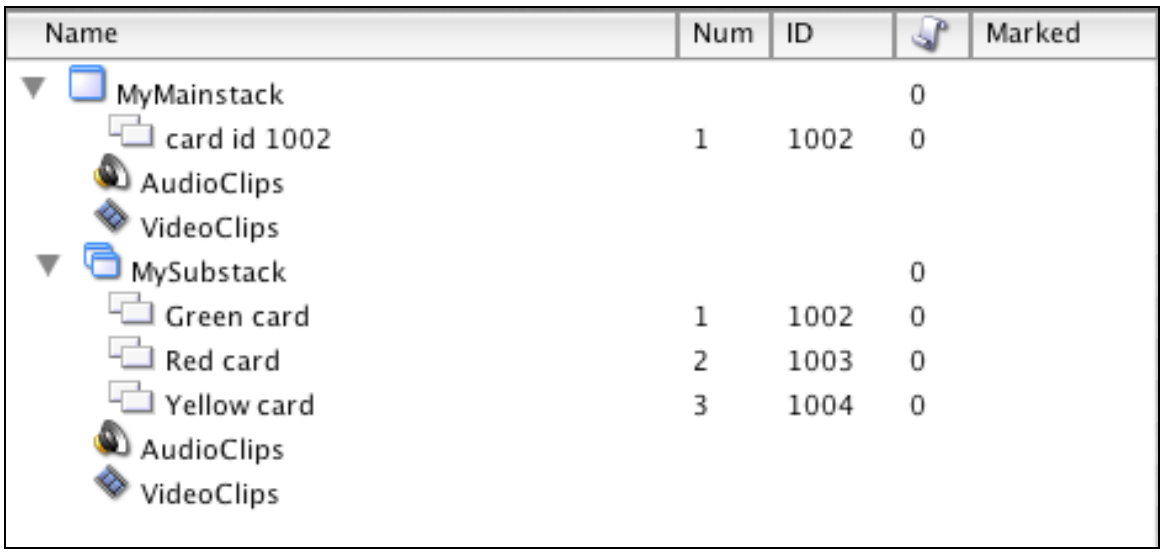
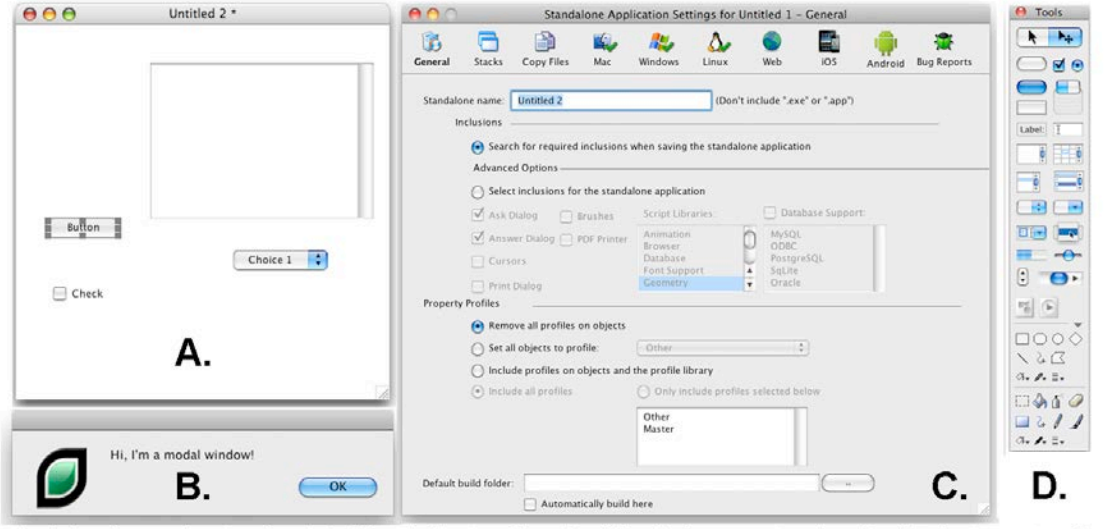


Fig. 2.5. Application Browser

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:

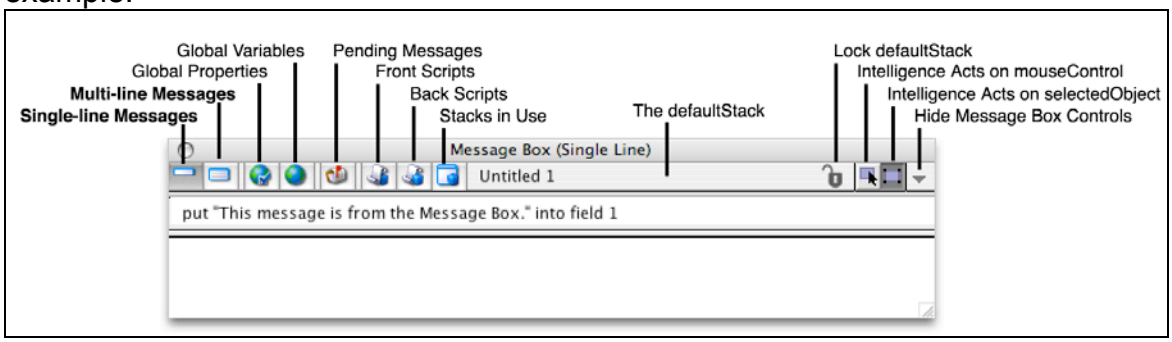


Fig. 2.7. Message Box

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.

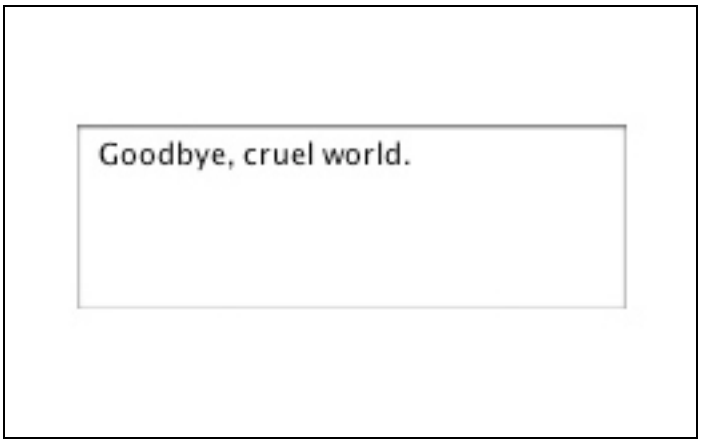


Fig. 2-8.

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 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.

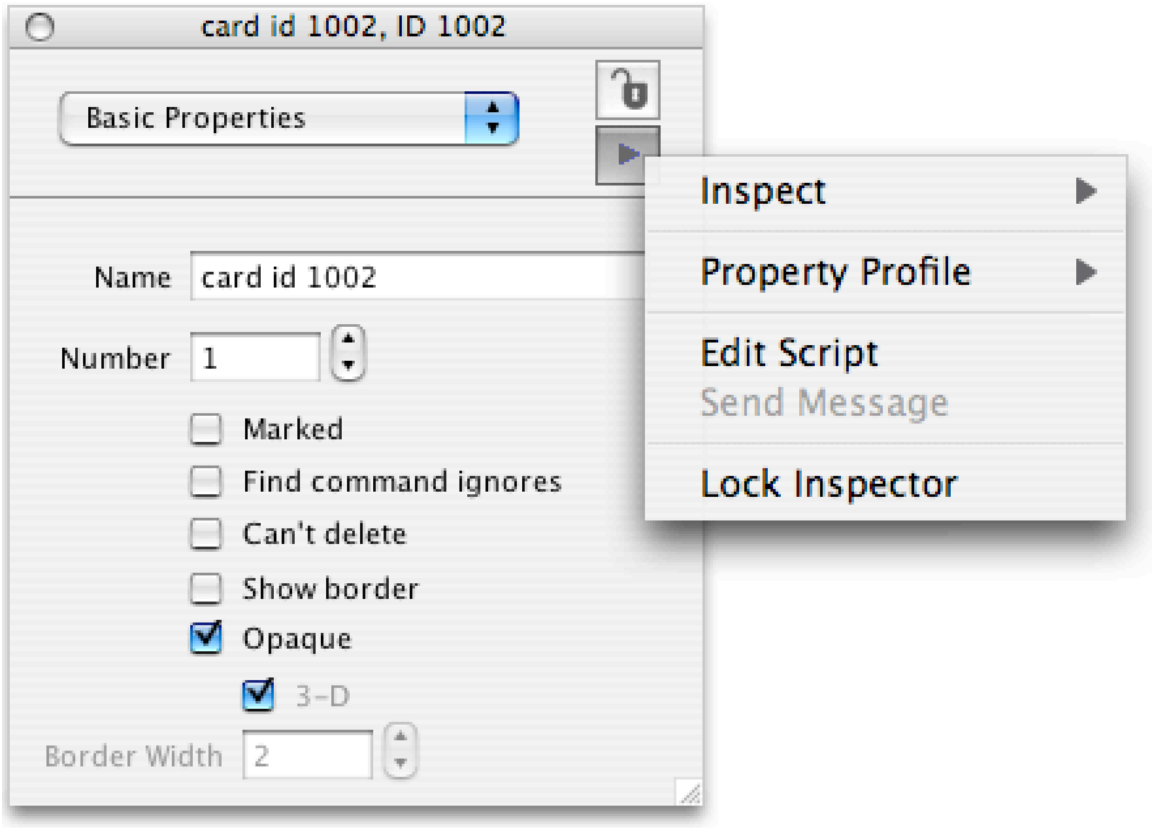


Fig. 3.1. Card Property Inspector

• 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.

I 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 hilite one button, the others in the **group** become unhilited. 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)

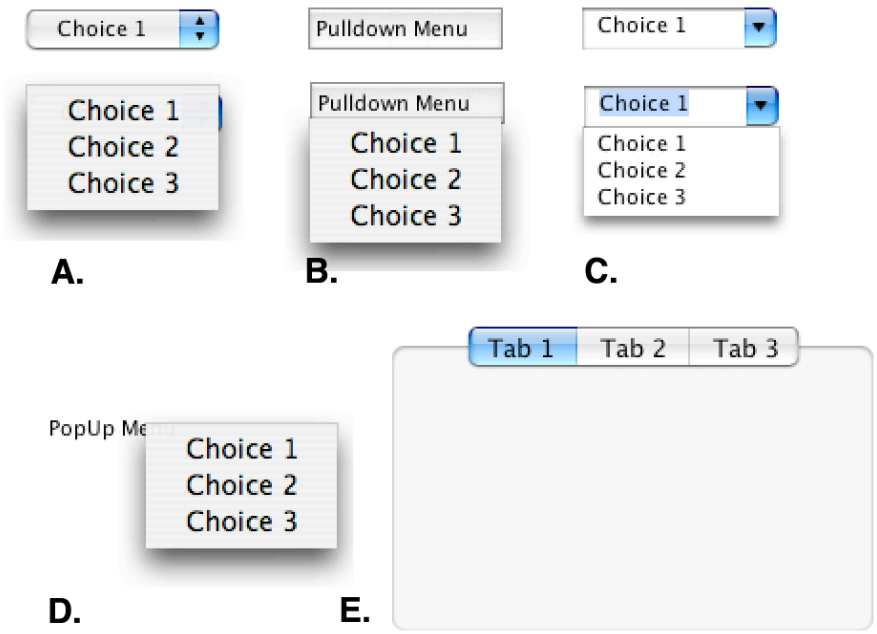


Fig. 3-2. Menu Objects.

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)

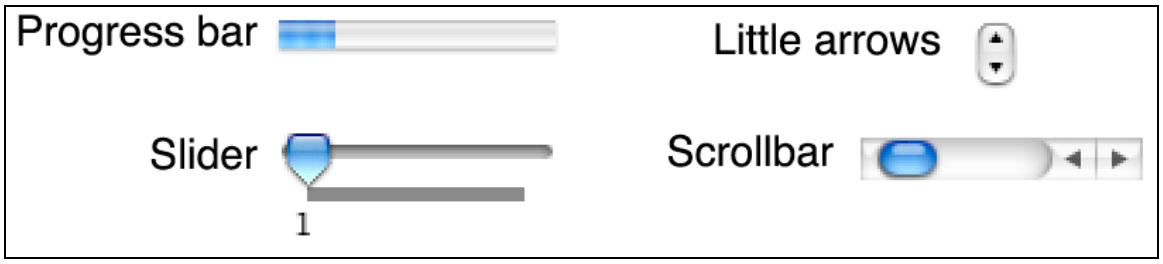


Fig. 3-3. Scrollbars

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)

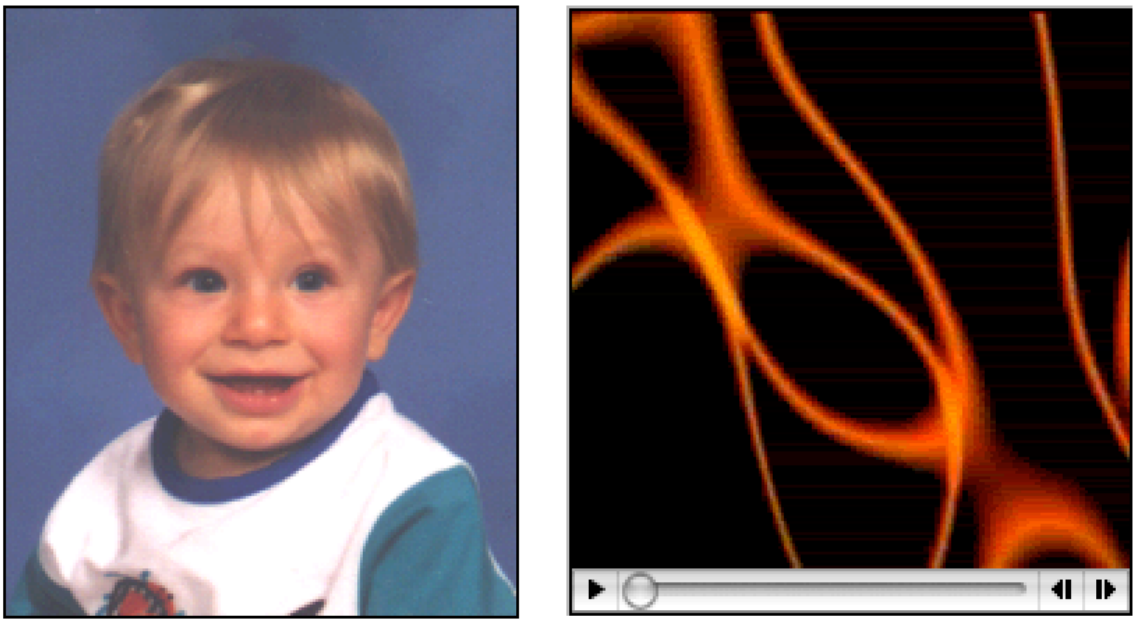


Fig. 3-4.

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**).

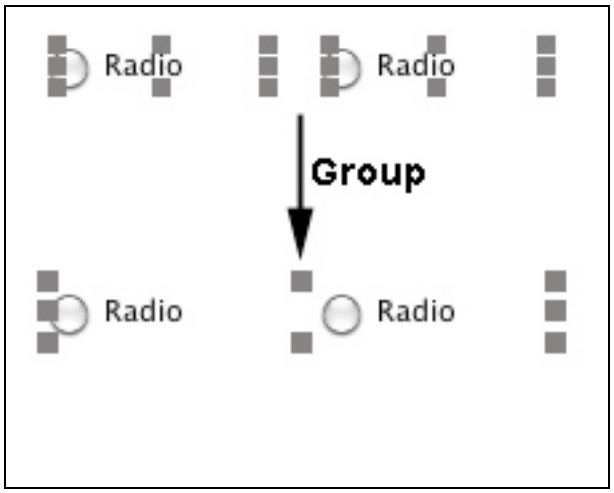


Fig. 4-1. Grouping.

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**.

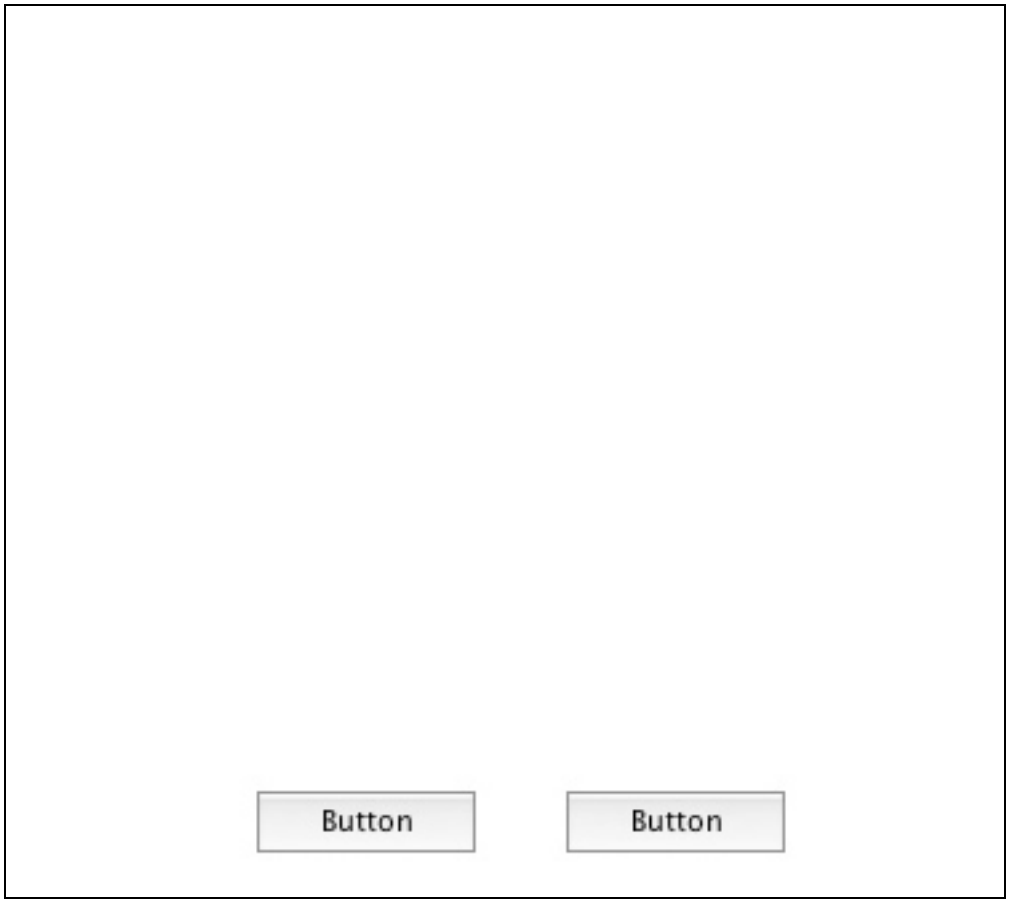


Fig. 4-2.

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.

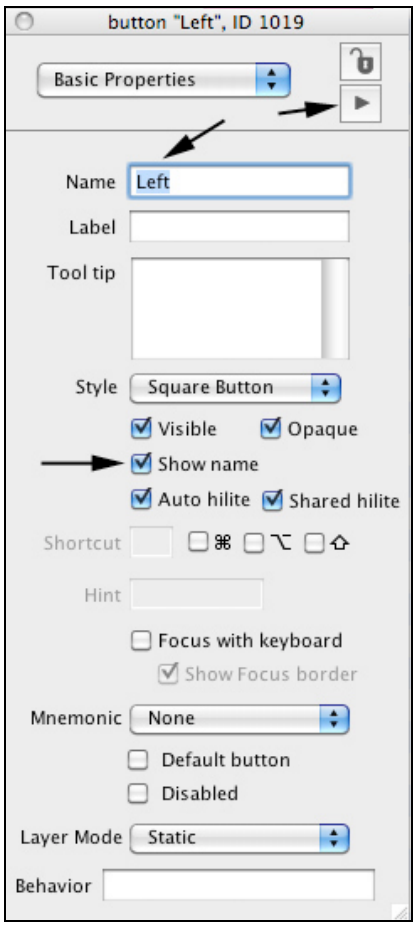


Fig. 4-3.

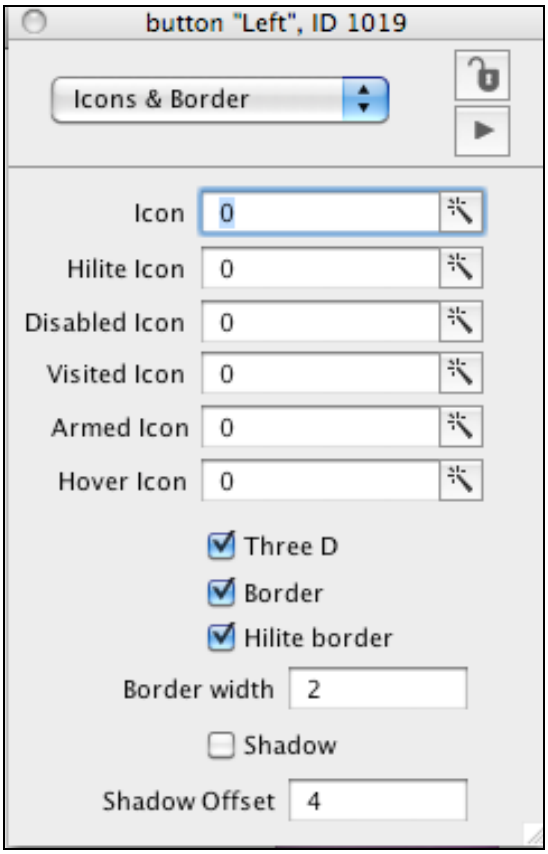


Fig. 4-4. Icons and Border.

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.

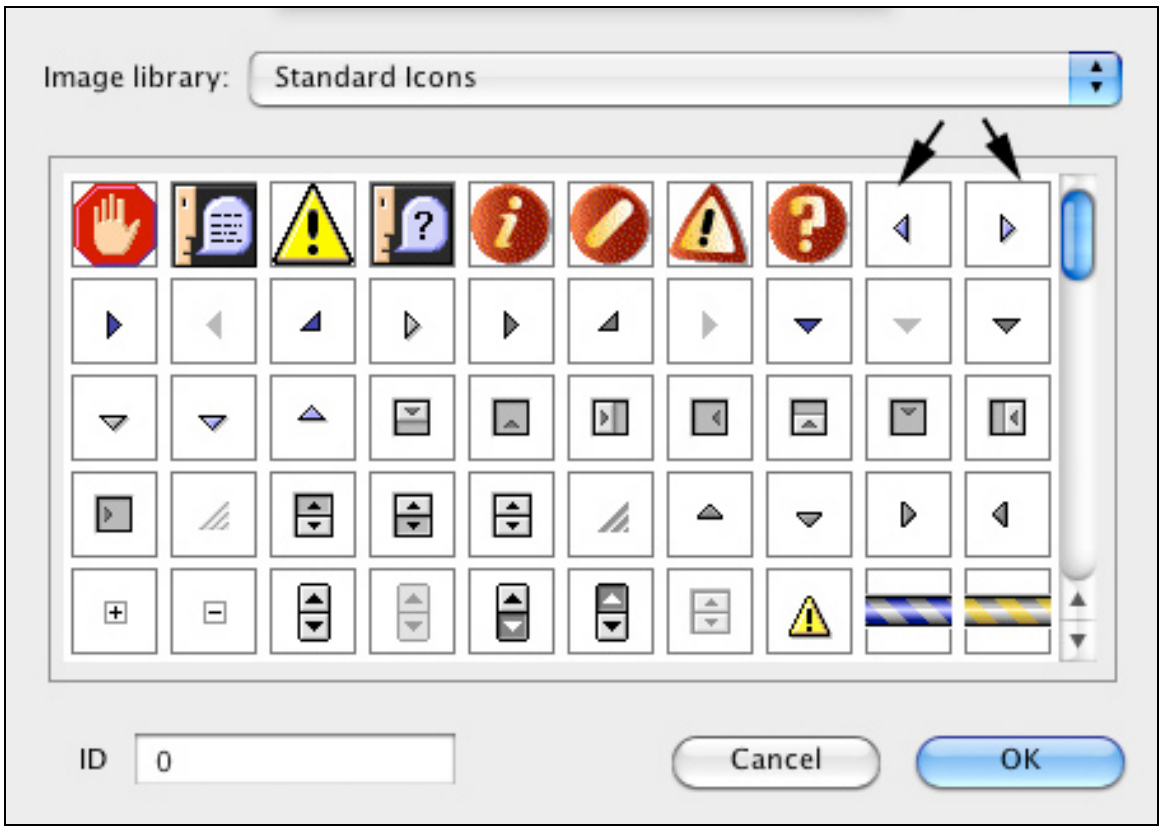


Fig. 4-5. Standard Icons.

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*).

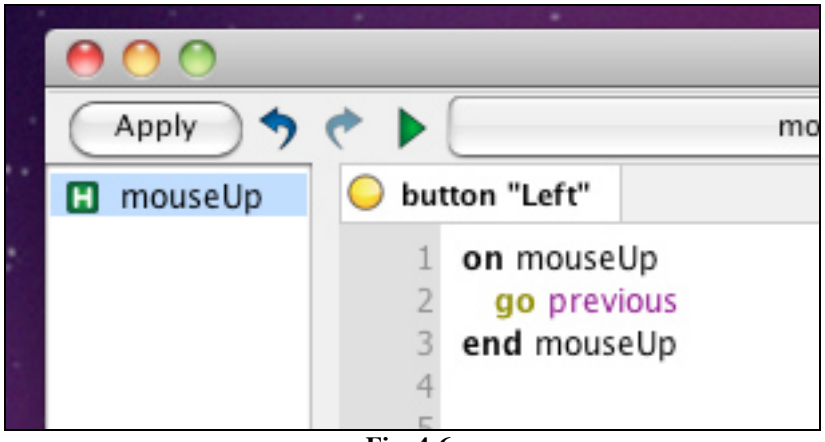


Fig. 4-6.

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.

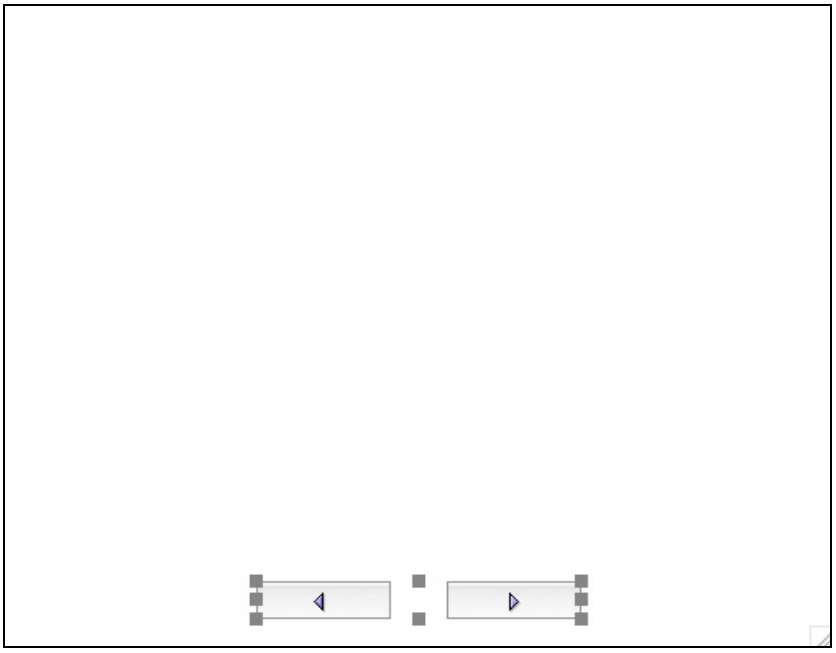


Fig. 4-7.

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.

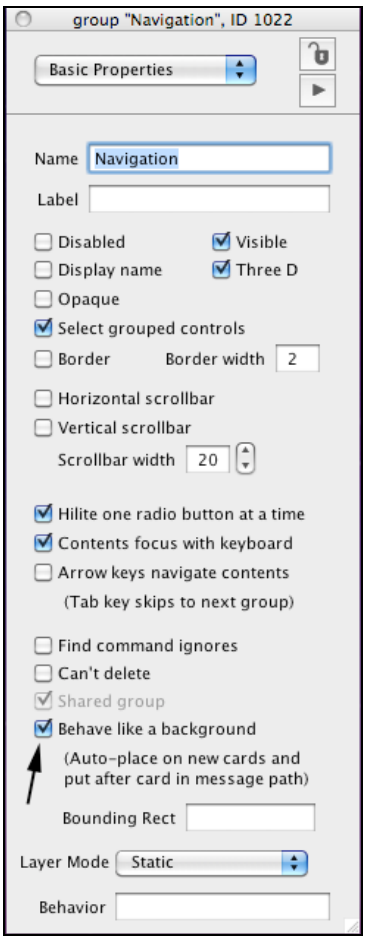


Fig. 4-8. Behave Like a Background.

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. It 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.

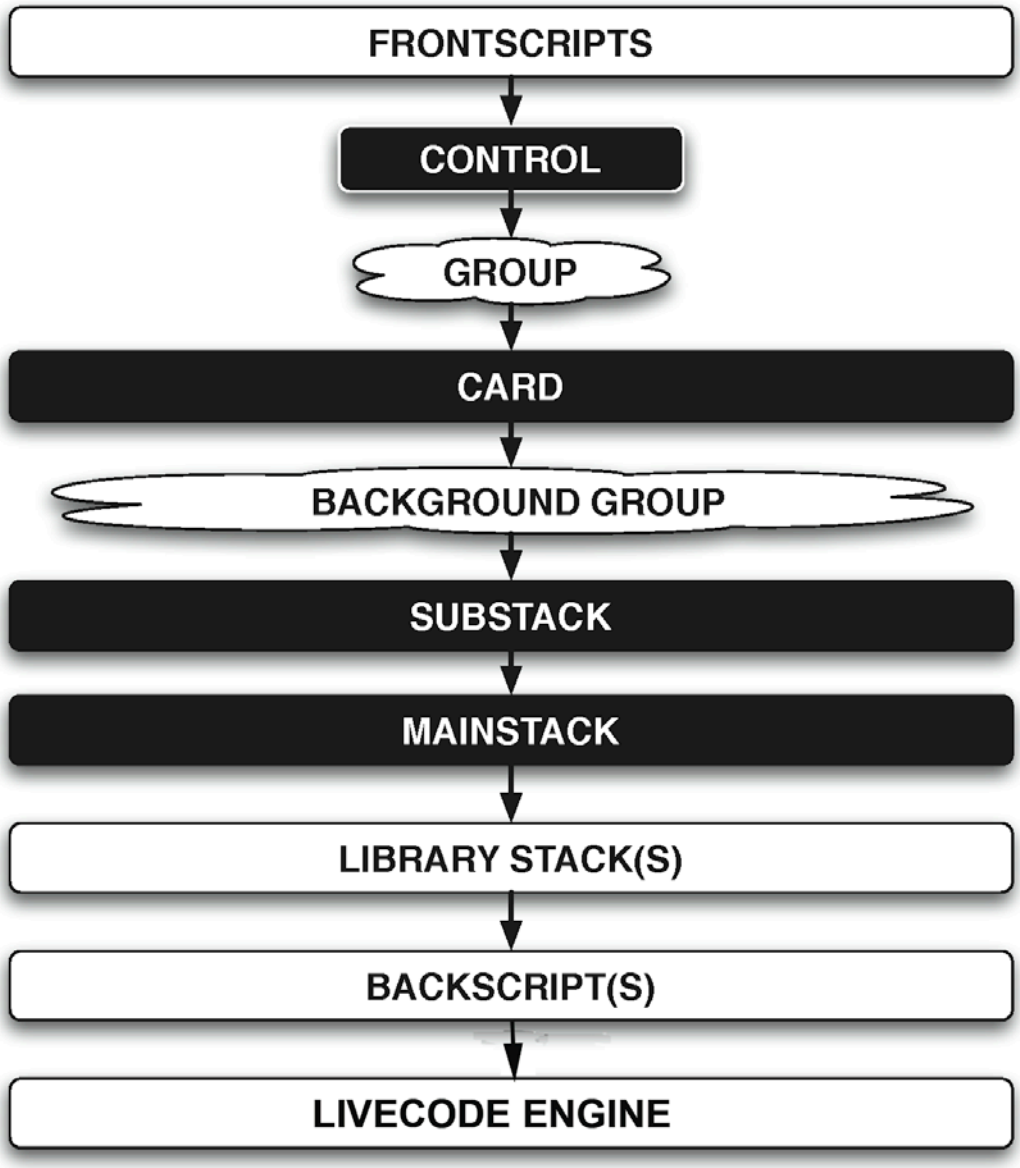


Fig. 6-1. LiveCode Message Hierarchy.

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!:

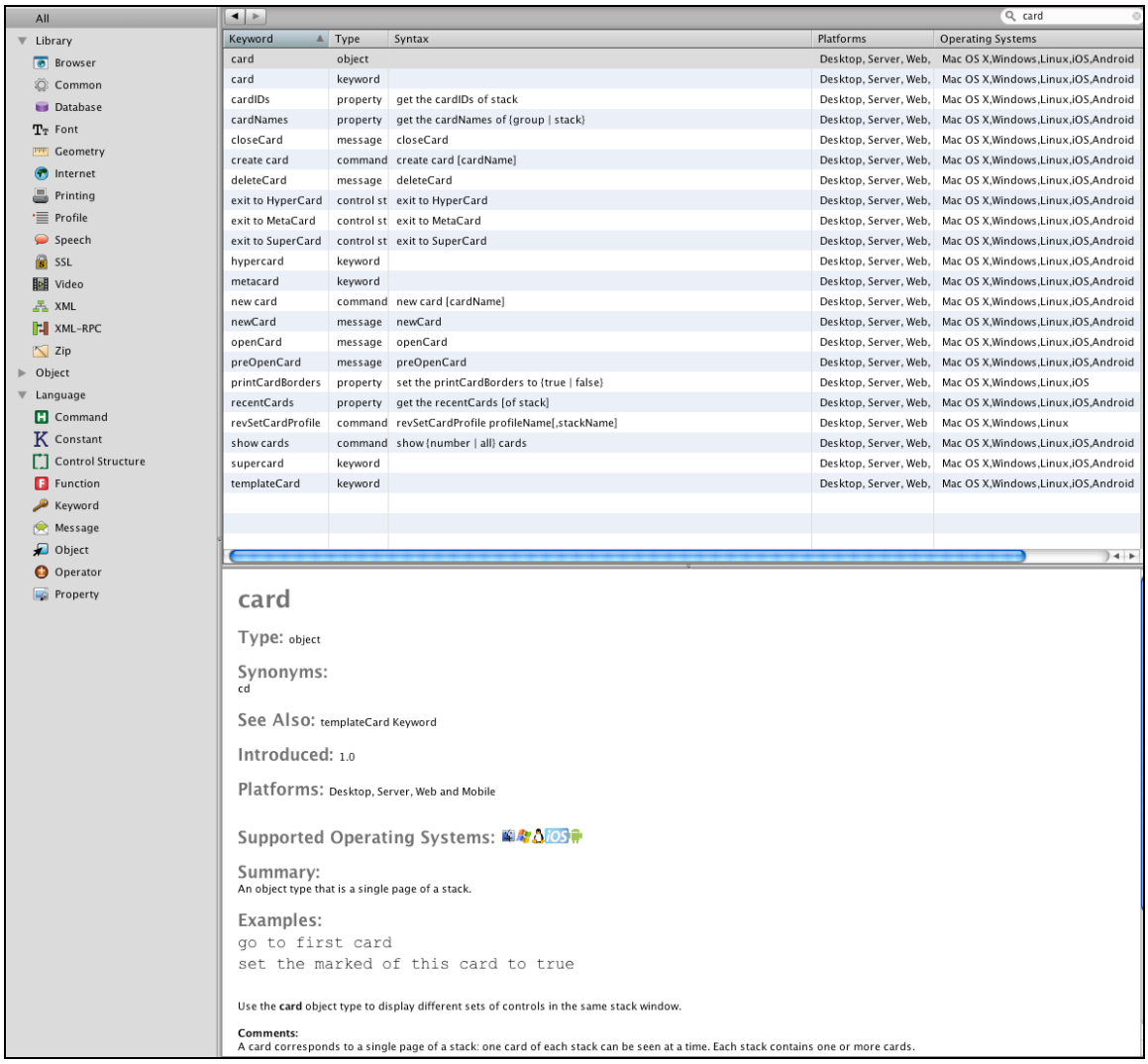
Be cautious in assigning scripts to background groups, particularly those with a mouseUp handler. It can lead to confusing results! A background group, however small visually, occupies the entire space behind the card (**Fig. 7-1**). Thus, by clicking on an empty area of the card, one may inadvertently 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.



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 make 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:

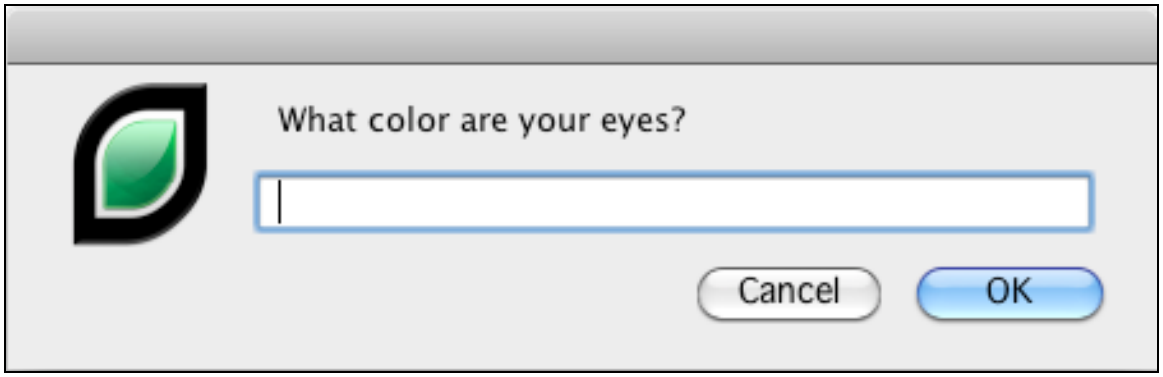


Fig. 9-2. Ask box.

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?"

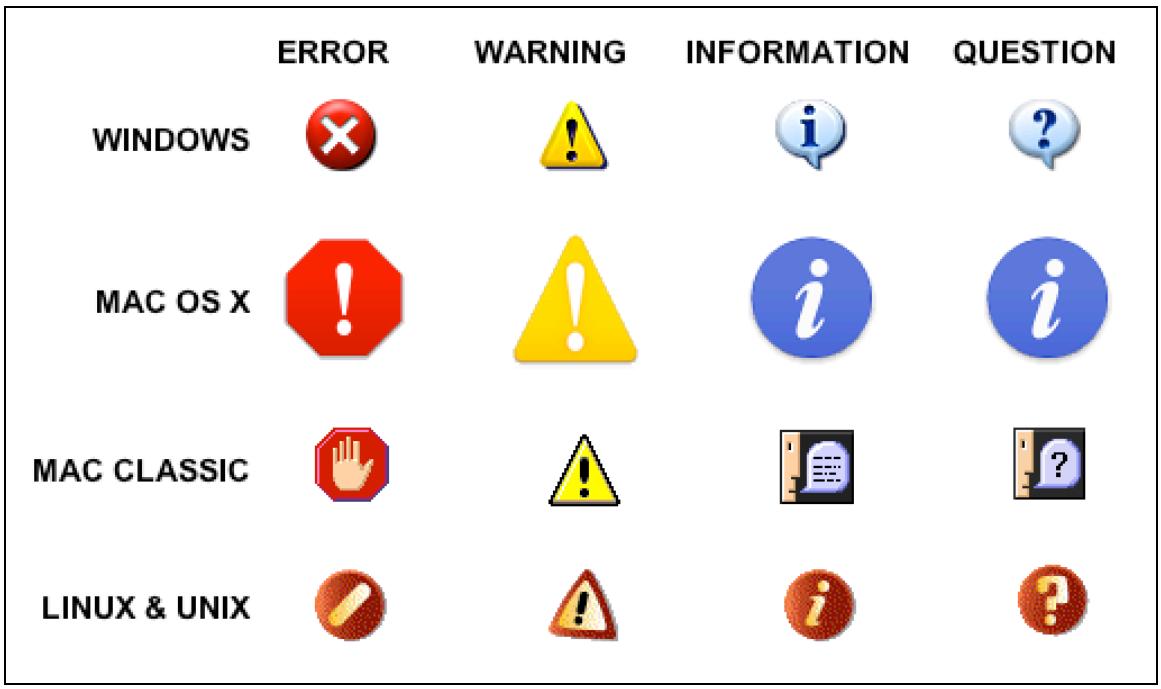


Fig. 9-3. Ask and Answer icons.

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.