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* an 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 in 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.

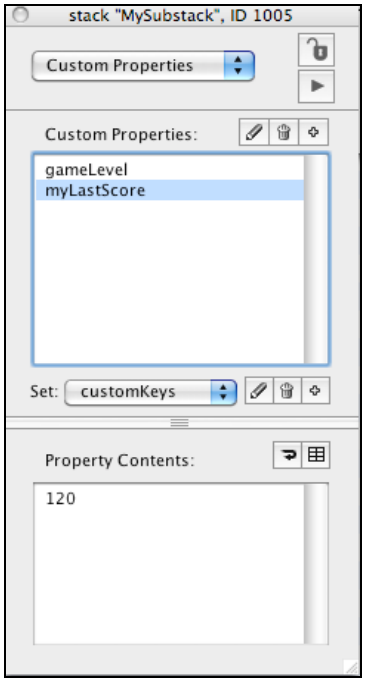


Fig. 11-1. Custom Properties

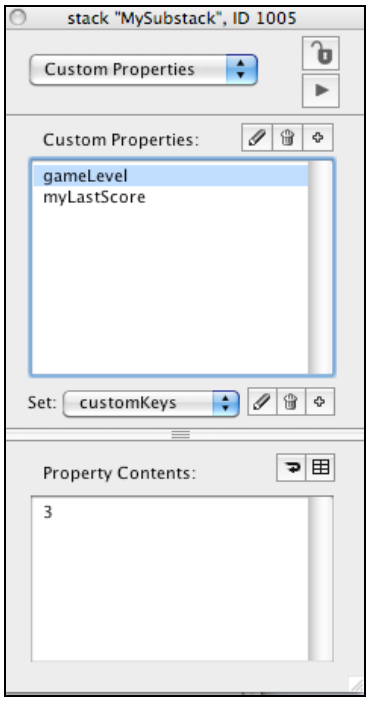


Fig. 11-2. Custom Properties

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 1/000 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 "overthecliff" into tname1
  put item 2 of line 2 of field "overthecliff" 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 imagination 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

T*he 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 specifiying 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 gobbledy 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 it 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.

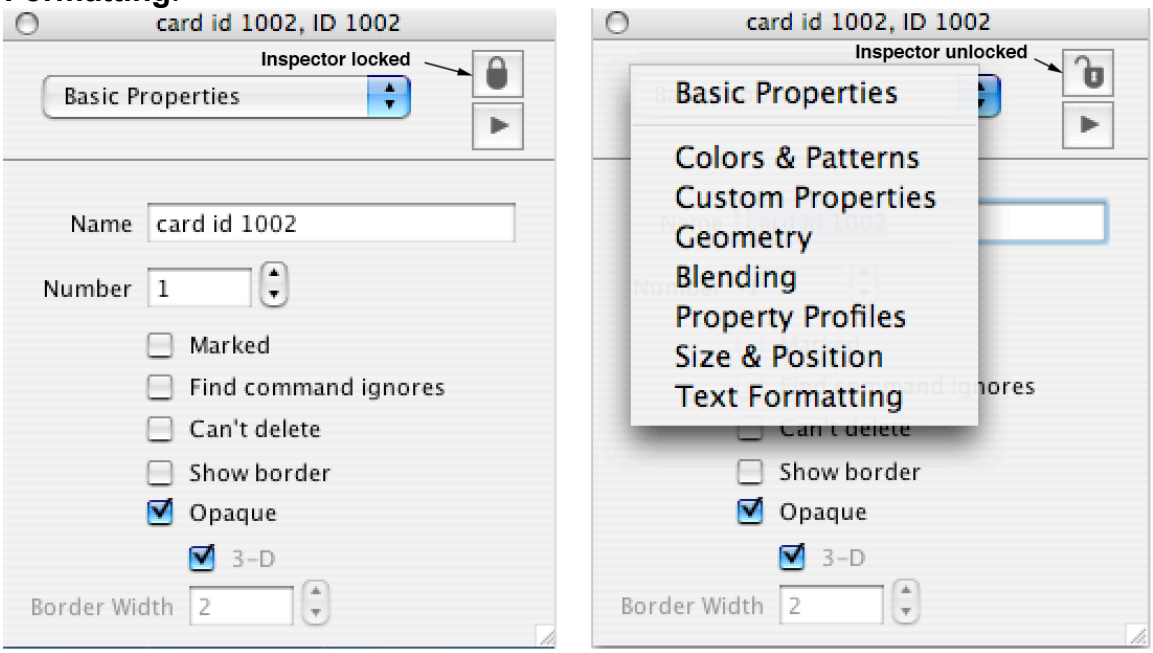
5. It may help to retype the script line in question; it may contain a hidden character, particularly if it was copied from a word-processing document.

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)

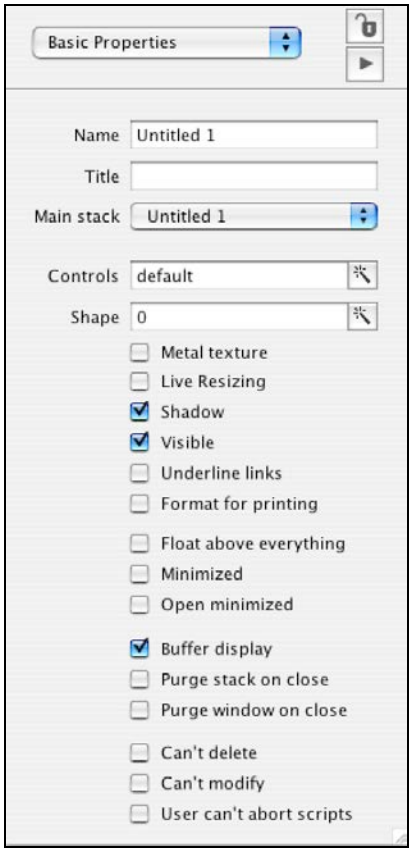


Fig. 22-1. Stack basic properties.

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)

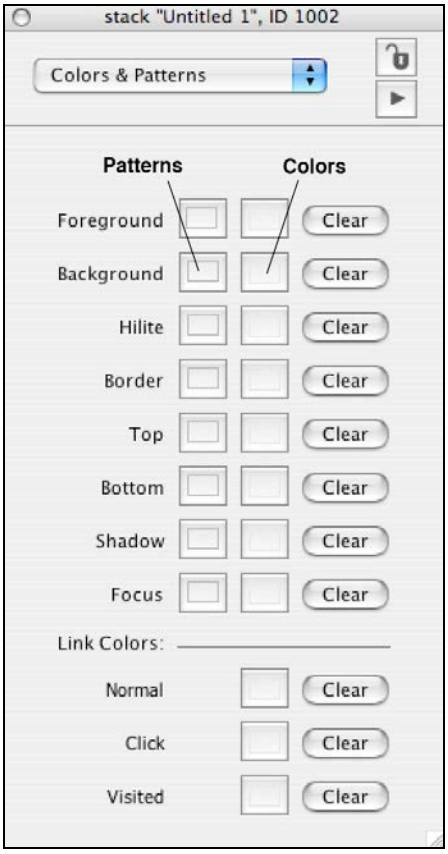


Fig. 22-3. Stack Colors and Patterns.

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.

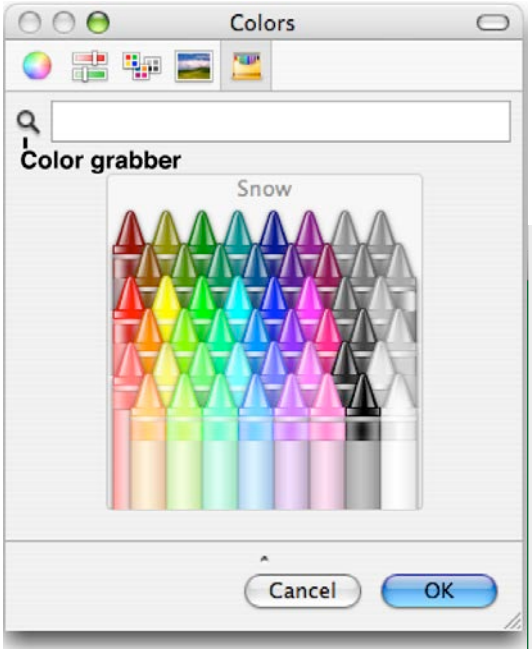


Fig. 22-4. Color grabber.

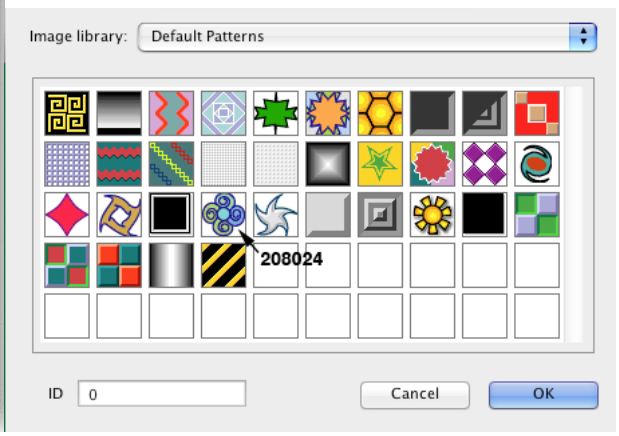


Fig. 22-5. Pattern IDs.

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 hiliting 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.

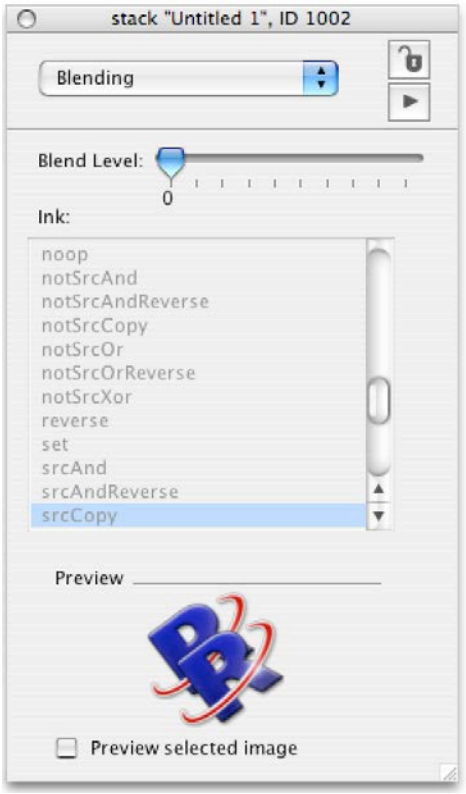


Fig. 22-6. Stack Blending.

STACK SIZE AND POSITION (Fig. 22-6)

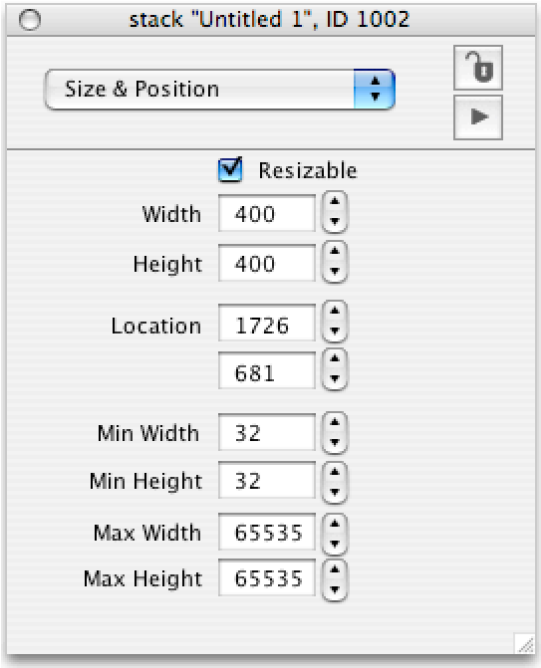


Fig. 22-7. Stack Size & Position.

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)

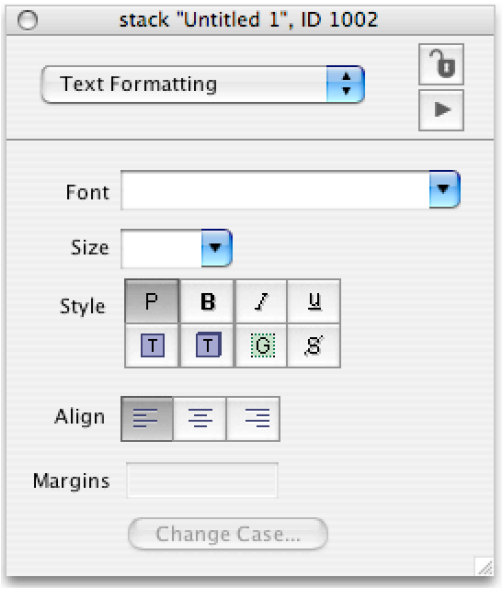


Fig. 22-8. Stack text formatting.

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)

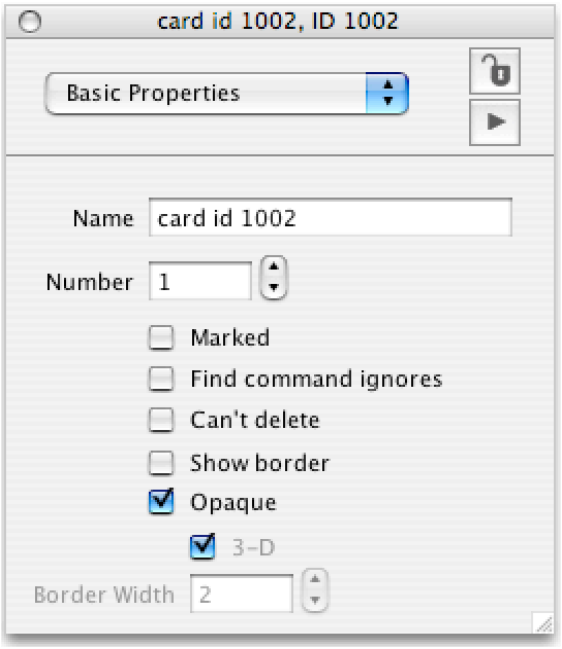


Fig. 23-1. Card Basic Properties.

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)

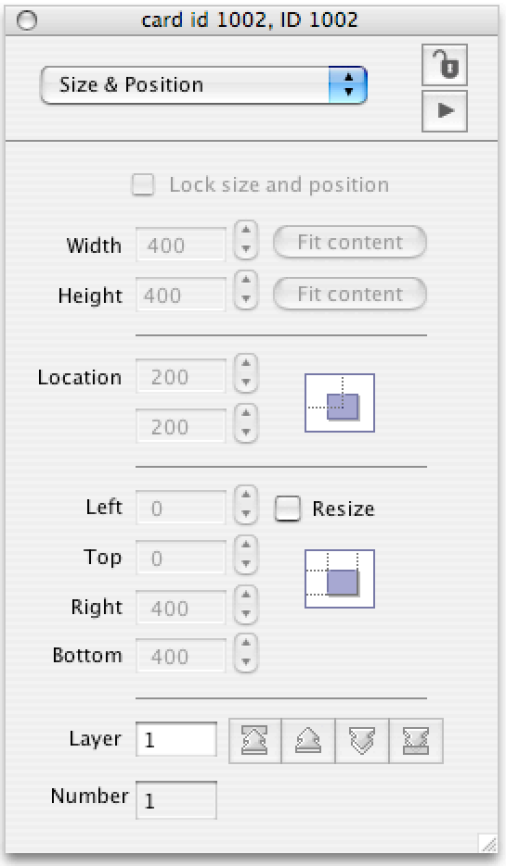


Fig. 23-2. Card Size and Position.

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

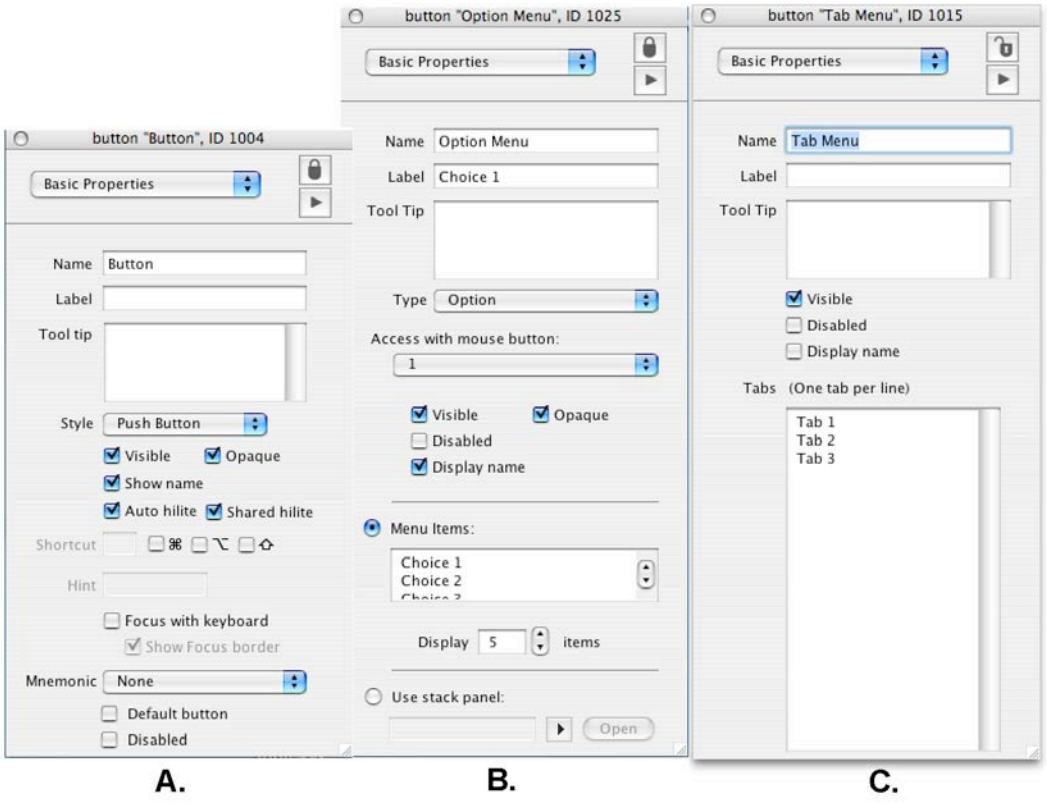


Fig. 24-1. Button Property Inspectors.

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)

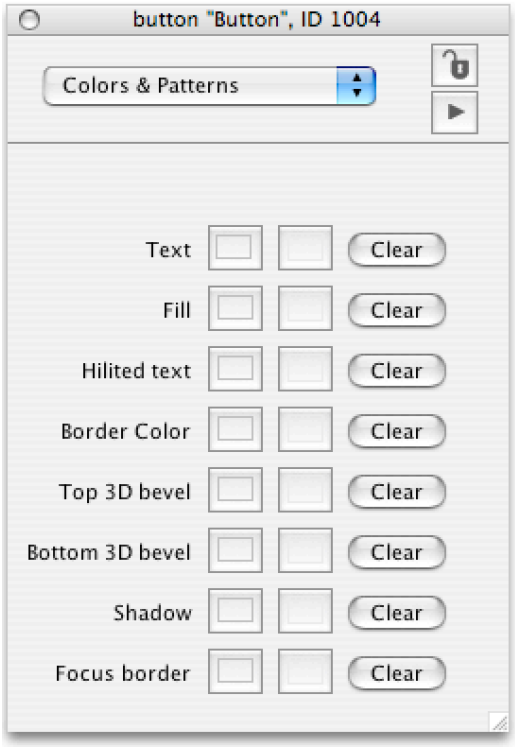


Fig. 24-3. Button Colors & Patterns.

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)