

Introduction

(<https://livecode.com/docs/9-5-0/introduction/>)

Lessons

(<https://livecode.com/docs/9-5-0/lessons/>)

FAQ (<https://livecode.com/docs/9-5-0/faq/>)

Language

(<https://livecode.com/docs/9-5-0/language/>)

Education Curriculum

(<https://livecode.com/docs/9-5-0/education-curriculum/>)

Deployment

(<https://livecode.com/docs/9-5-0/deployment/>)

Components

(<https://livecode.com/docs/9-5-0/components/>)

Tooling

(<https://livecode.com/docs/9-5-0/tooling/>)

Core Concepts

(<https://livecode.com/docs/9-5-0/core-concepts/>)

Language Comparison

(<https://livecode.com/docs/9-5-0/language-comparison/>)

Extending LiveCode

(<https://livecode.com/docs/9-5-0/extending-livecode/>)

Extending LiveCode

(<https://livecode.com/docs/9-5-0/extending-livecode/>)



Improve this documentation

(<https://github.com/livecode/livecode/blob/develop-9.5/docs/guides/LiveCode%20Builder%20Style%20Guide.md>)

LiveCode Builder Style Guide

Introduction

This document describes best practices to be followed when working with LiveCode Builder source code. Please follow it *especially* when writing code to be included with the main LiveCode product package.

Copyright headers

Please include a license header at the top of the

`.lcb` file.

For the main LiveCode repository, or for any community extensions, the license is the GNU General Public License v3

Introduction

headers

Naming

Documenti
the source
code

Named
constants

Whitespac

Widget-
specific
guidelines

0/extending-livecode/extending-livecode/)

LiveCode Builder Language
Reference

(<https://livecode.com/docs/9-5-0/extending-livecode/livecode-builder-language-reference/>)

LiveCode Builder Style Guide

(<https://livecode.com/docs/9-5-0/extending-livecode/livecode-builder-style-guide/>)

LiveCode Builder Style Guide

Whats New?

(<https://livecode.com/docs/9-5-0/whats-new/>)

(<http://www.gnu.org/licenses>)

without the "any later
version" clause.

Naming

Module name

The module name uses reverse DNS notation. For example, a module created by the Example Organization would use module names beginning with

```
org.example .
```

Replace any hyphen (-) characters in a domain name with underscore (_)

characters. For example, a module derived from the

```
fizz-buzz.example.org
```

domain could be

```
org.example.fizz_buzz .
```

Additionally, add an underscore (_) to the start of any element in the domain name starting with a digit.

For example, a module derived from the

```
999.example.org
```

could be

```
org.example._999 .
```

You must only use module names corresponding to domain names that you control or are allowed to use.

This restriction is enforced by the the LiveCode extension store.

If you don't have a domain name of your own, you may use module names beginning with

```
community.livecode.  
<username>
```

, replacing `<username>` with the username you use to log into the LiveCode extension store. For example, if your username is "sophie", then you can create a module named

```
community.livecode.sophie.mymodule
```

.

For the main LiveCode repository, please use module names beginning with `com.livecode` .

Always write module names in lower case.

Naming variables and parameters

Give variables and parameters

```
xCamelCaseNames
```

 with a

leading lowercase character indicating their scope and availability.

The meanings of the leading lowercase characters are:

Prefix	Context	Meaning
k	all	constant
s	module	static variable
m	widget	static variable
p	handler definitions	in argument
r	handler definitions	out argument
x	handler definitions	inout argument
t	handler bodies	local variable

In general, please use nouns to name your variables and parameters. Make the names descriptive; for example:

```

variable
tOutputPath as
String -- Good
variable tString
as String --
Bad

```

For `Boolean` variables, please try to use "yes or no" names. For example:

```
variable  
tIsVisible as  
Boolean  
variable  
tHasContents as  
Boolean
```

Naming handlers

Give handlers `camelCase` names.

Use verbs to name your handlers. For example,

```
handler  
rotateShape(inout  
xShape, in  
pAngleInDegrees)  
-- ...  
end handler
```

Naming types

Give types `TitleCase` names.

To distinguish from handlers, use nouns to name your types. For example,

```
type RotationAngle  
is Number
```

Documenting the source code

In-line documentation for a definition is extracted from a `/** */` comment block immediately before the start of the definition.

Always add a top-level documentation block at the start of the LCB file describing your widget, library or module. In addition, add in-line documentation to all of the following:

- `syntax`

definitions

- `property`

definitions

- `public handler`

definitions in libraries and modules

- `public variable`

definitions in modules

It is particularly important to add documentation to all syntax and to any public handlers that aren't primarily accessed using syntax.

Additionally, add documentation for all messages that are posted by a widget. The documentation for each message must be

placed in the top-level documentation block for the widget. For example:

```
/*  
The navigation  
bar widget is  
intended for use  
in mobile apps  
for  
switching between  
cards, although  
there are many  
other possible  
uses.  
  
...  
  
Name:  
hiliteChanged  
Type: message  
Syntax: on  
hiliteChanged  
Summary: Sent  
when a navigation  
item is selected  
  
...  
*/  
widget  
com.livecode.widget  
.navbar  
-- ..  
end widget
```

Please refer to the [Extending LiveCode](Extending LiveCode.md) guide for full details of the syntax of in-line documentation comments, including examples.

Named constants

Often, it is useful to use constant values in your code. Please declare named constants rather than placing the values in-line. For example, you may want to create three tabs labelled "Things", "Stuff", and "Misc":

```
constant kTabNames
is ["Things",
   "Stuff", "Misc"]

handler
createTabs()
    variable tName
    repeat for each
element tName in
kTabNames
    -- Create
the tab
    end repeat
end handler
```

In particular, please avoid any "magic numbers" in your code.

Whitespace

Indentation

Please indent with tab characters. Use one tab character per level of indentation.

Please do not use a level of indentation at `module` level.

Comments should be indented to the same level as the code they apply to.

For example:

```
module
org.example.indent

-- Example
handler
handler fizzBuzz(in
pIsFizz)
    if pIsFizz then
        return
        "Fizz"
    else
        -- Maybe
        this should have
        a capital letter
        return
        "buzz"
    end if
end handler

end module
```

If it's necessary to mix spaces and tabs for indentation, please use 3 spaces per tab.

Wrapping

Avoid lines longer than 80 characters. Break long lines using a `␣` continuation character. Indent continuation lines by two levels. For example:

```
constant kWordList
is ["a", "very",
    "long", "list",
    "that", "is",
    "much",
        "more",
    "readable", "when",
    "wrapped",
    "nicely"]
```

When breaking a handler definition or handler type definition, break long lines at commas:

```
handler
processStringAndArray(in pStringArg as
String,
    in pArrayArg
as Array) returns
Boolean
```

Handler declarations, definitions and calls

In handler definitions and handler type definitions, don't insert whitespace between the handler name and the parameter list. For example:

```
handler type
Fizzer()  -- Good
handler type Buzzer
()  -- Bad
```

In handler parameter lists,
please add a space between
each parameter. For
example:

```
handler
formatAsString(in
pValue, out
rFormat) -- Good
handler
isEqualTo(in
pLeft,in pRight)
-- Bad
```

Please observe the same
formatting in handler calls.
For example:

```
variable
tFormatted
variable tIsEqual
formatAsString(3.14
15, tFormatted)
-- Good
isEqualTo
(tFormatted,"3.1415
") into tIsEqual -
- Bad
```

List and array literals

In list and array literals,
please add a space between
each element, after the
comma. For array literals,
also insert space between
key and value, after the
colon. For example:

```
constant
kPowersOfTwo is [1,
2, 4, 8]
constant
kBorderWidths is
{"top": 0,
"bottom": 0,
"left": 1, "right":
2}
```

Widget-specific guidelines

This section contains recommendations that are specific to writing widgets.

As a general rule, try to minimize surprise for users who mix widgets with classic LiveCode controls by using similarly named events and properties with similar semantics.

Writing load handlers

When writing an `onLoad()` handler to initialise a widget from serialised state:

- Always call property setters to update the widget state. Do not set instance variables directly.
- Always check that the property array contains each key rather than

accessing it unilaterally.

- If keys are absent from the property array, do not set them to default values. Rely on the

```
onCreate()
```

handler to have already done that.

Example:

```
public handler
onLoad(in
pProperties)
    if "showBorder"
is among the keys
of pProperties then

    setShowBorder(pProp
erties["showBorder"
])
    end if
    if "opaque" is
among the keys of
pProperties then

    setShowBackground(p
Properties["opaque"
])
    end if
end handler
```

Classic control properties

Where possible, try to make the names and effects of widget properties as similar as possible to properties of classic controls. For example:

- **label**: the text for the primary text label
- **showBorder**: whether to display an outline
- **opaque**: whether to fill in the background

Host control properties

When relying on properties that are implemented by the widget host control, such as the classic color and pattern properties, include `metadata` definitions to display them in the PI, but do not include a `property` definition.

Because there is no `property` definition, the documentation for the host control property must be placed in the top-level documentation block comment before the start of widget definition.

For example, to shadow the `borderColor` host control property:

```
/**
 * Test widget that
 * demonstrates
 * shadowing the
 * <borderColor(property)> property.
 * ...
```

Name: borderColor
Type: property

Syntax: set the
borderColor of
<widget> to <color>
Syntax: get the
borderColor of
<widget>

Summary: Color for
drawing circular
outline

Description:
The <borderColor>
property controls
the color used to
draw the
widget's outline.
*/
widget
org.example.host_co
ntrol_property

-- ...

```
metadata
borderColor.editor
is
"com.livecode.pi.co
lor
metadata
borderColor.section
is "Colors"
metadata
borderColor.label
is "Border color"
metadata
borderColor.default
is "109,109,109"
```

-- ...

end widget

Events

Always implement:

- onCreate()

- onSave()
- onLoad()
- onPaint()

You should *usually* implement:

- onGeometryChanged()

: if you have any non-trivial recomputation required to handle resizing

- onMouseDown()

/

onMouseUp()

/

onMouseMove()

: always

post

these to script in order to behave like a classic control

Offline (Leave a message)