

LiveCode 9.6.4 User Guide

- LiveCode Script
 - Introduction
 - Welcome
 - Where to Begin
 - System Requirements
 - All Operating Systems
 - Requirements for Windows Systems
 - Requirements for Linux Systems
 - Requirements for Mac OS X Systems
 - Using the Documentation
 - Documentation Conventions
 - Menu Reference
 - Keyboard Shortcuts
 - Screen Snapshots
 - Navigating the Documentation
 - Start Center
 - Social Media Links
 - Recent Stacks
 - Quick Links
 - Dictionary
 - LiveCode APIs
 - LiveCode User Guides
 - RevOnline
 - Additional Resources
 - Developer Community
 - Technical Support
 - Getting Started
 - Prerequisites
 - Event Driven Programming
 - Object-Based Programming
 - The 'Edit' and 'Run' Mode
 - Structuring your Application
 - Cards, Stacks & Files
 - The Structure of a Stack File
 - Opening a Stack File
 - Main Stacks and Substacks
 - Stacks, Stack Files, and Memory
 - Media & Resources
 - How to Use External Files
 - When to use a Database

- LiveCode Memory Limits
 - A note about entries designated as "Unlimited":
- Building a User Interface
 - Creating and Organizing Objects
 - Creating Controls with the Tools Palette
 - Alignment & Layering
 - The Keyboard Focus
 - Object & Control Types
 - Stacks – for displaying windows, palettes and dialog boxes
 - Window Types and the Mode of a Stack
 - The Standard Window Types
 - Editable Windows – for documents
 - Modeless Dialog Boxes – for alerts and settings
 - Modal Dialog Boxes – for alerts and settings
 - Palettes – for accessory and tool windows
 - Ask Question Dialog – for asking a question
 - Answer Alert Dialog – for displaying a dialog
 - File Selector Dialogs
 - Color Chooser Dialog
 - Printer Dialogs
 - Alpha Blend Windows – for Enhanced Tooltips and Multimedia
 - System Palettes – for utilities floating above applications
 - Stack menus – for displaying non-standard menus
 - Stack Decorations – for Window Appearance
 - Button Controls – for performing actions
 - Text Field Controls – for displaying or entering text
 - List and Table Field Controls
 - Table Field Control – for displaying a table
 - Data Grid Control – for presenting data
 - Cards
 - Groups & Backgrounds – for organizing, and sharing controls
 - Graphics, Images, Players, Audio & Video Clip Objects – for multimedia
 - Menu Controls – for displaying choices
 - Other Controls
 - Using the Menu Builder
 - Using the Geometry Manager
 - 10 Tips for Good User Interface Design
- Coding in LiveCode
 - The Structure of a Script
 - What is a Script
 - The Types of Handler
 - Message Handlers
 - Function Handlers
 - GetProp Handlers
 - SetProp Handlers
 - Comments
 - Compiling a Script

- Summary
- Events
 - What Causes Messages to be Sent
 - Responding to Events
- The Message Path
 - The Object Hierarchy
 - The Message Path
 - The Message Target
 - Handlers with the Same Name
 - Trapping Messages
 - Blocking System Messages
 - Passing a Message to the Next Object
 - Selectively Trapping or Passing Messages
 - Groups, Backgrounds & The Message Path
- Commands and Functions
 - Using Built-in Commands and functions
 - Commands
 - Functions
 - Writing function calls with the "the" form
 - Custom Commands and Functions
 - Using custom commands
 - Using custom functions
 - Passing Parameters
 - Passing Multiple Parameters
 - Parameter Variables
 - Parameter variable names
 - Empty parameters
 - Setting a default value for a parameter
 - Implicit Parameters
 - Passing Parameters by Reference
 - Empty parameters
 - Returning Values
 - Returning an Error from a Message Handler
 - Summary
- Variables
 - Variable Scope
 - Local Variables
 - Script Local Variables
 - Global Variables
 - Variable Names
 - Special Variable Types
 - Parameter variables

- Environment variables
 - Command-line argument variables
 - The special variable "it"
- Array Variables
 - Deleting Elements of an Array
 - Listing the Elements in an Array
 - Listing Nested Elements within an Element
 - Transforming a List of Data into an Array
 - Combining the Elements of an Array into a List
 - Nesting an Array
 - More Information
- Constants
 - Built-in constants
 - User-defined constants
- Containers, Operators & Sources of Value
 - What is a Container?
 - Setting and Retrieving Data from Containers
 - What are Sources of Value?
 - Getting and Setting Properties
 - What are Literal Strings?
 - Using Literal Strings
 - Quoting strings
 - What are Operators?
 - Numeric Operators
 - String operators
 - Logical Operators
 - Binary versus Unary Operators
 - Conversion of Values
 - Operator Precedence
 - Using the Grouping Operator ()
 - Factors and Expressions
 - Defining Factors
 - When it matters
- Making Decisions
 - If ... then ... else
 - Switch
- Extending the Message Path
 - Creating a Code Library
 - Using backScripts
 - Using frontScripts
 - Using a Stack's Script with *start using*
 - Sending Messages Directly to Objects
 - The *send* Command versus the *call* Command

- Writing reusable code using behaviors
- Timer Based Messaging
 - Delivering a Message in the Future
 - Repeating a Timer Message
 - Canceling a Timer Message
 - Displaying a List of Pending Timer Messages
- Tips for Writing Good Code
 - Variable naming
 - Variable scope
 - Use of comments
 - Use of functions
 - Explicit Variables
- Programming a User Interface
 - Introduction
 - Referring to Objects
 - Referring to objects by name
 - Referring to objects by number
 - Referring to objects by ID
 - Referring to objects by ordinal
 - The special descriptor 'this'
 - Control references
 - Nested Object References
 - Avoid using numbers as object names
 - Properties
 - Using Object Properties
 - Referring to properties
 - Changing properties
 - Property Inheritance
 - Overriding inheritance
 - Global Properties
 - Referring to global properties
 - Changing global properties
 - Saving and restoring global properties
 - Text Related Properties
 - Text style properties
 - Formatted text properties
 - Creating and Deleting Objects
 - The Create Object Command
 - The Delete Object Command
 - Creating Objects Off-screen Using Template Objects
 - Property Arrays using the Properties Property
 - Custom Properties

- Using Custom Properties
- Creating a Custom Property
- The Content of a Custom Property
- Custom Property Names
- Referring to Custom Properties
- Nonexistent Custom Properties
- Finding out Whether a Custom Property Exists
- Custom Properties & Converting Text Between Platforms
- Storing a file in a custom property
- Deleting a custom property
- Custom Property Sets
 - Creating custom property sets
 - Custom property set names
 - Referring to custom property sets
 - Finding out whether a custom property set exists
 - The default custom property set
 - Using multiple custom property sets
 - Copying custom properties between property sets
 - Arrays, custom properties, and custom property sets
 - Deleting a custom property set
- Attaching Handlers to Custom Properties
 - Responding to changing a custom property
 - The structure of a *setProp* handler
 - Responding to a request for the value of a custom property
- Virtual Properties
 - When to use virtual properties
 - Handlers for a virtual property
 - Creating new object properties
- Property Profiles
 - Profile names
 - The master profile
 - Switching between profiles
 - Creating a profile in a handler
 - Adding profile settings in a handler
- Managing Windows, Palettes and Dialogs
 - Moving a window
 - Changing a window's layer
 - The active window
 - Creating a backdrop
 - Open, Closed, and Hidden Windows
 - The states of a stack
 - **Window Types and the Mode Property**
 - Window Appearance
 - The Decorations Property
 - Making a stack resizable
 - Irregularly-Shaped and Translucent Windows

- Programming Menus & Menu Bars
 - Menu Types
 - Button Menus
 - Creating Cascading Menus
 - Ticks, Dashes & Checks in Menus
 - Menu Bars on Linux and Windows Systems
 - Menu Bars on Mac OS X Systems
 - Stack Menus
 - Displaying Context Sensitive Menus
- Searching and Navigating Cards using the Find Command
- Using Drag and Drop
 - Initiating a Drag Drop
 - Tracking During a Drag Drop Operation
 - Responding to a Drag and Drop
 - Prevent Dragging and Dropping to a Field
- Error Handling & Debugging
 - Introduction
 - Common Techniques for Solving Problems
 - Errors During Execution
 - Errors During Compiling
 - Suppressing Errors and Messages
 - Outputting information as your script executes
 - Interrupting Execution
 - The Debugger
 - Custom Error handling
 - The Message Watcher
 - Tracking Down Problems in Standalone Applications
- LiveCode Data Grid
 - Introduction
 - What is the Data Grid?
 - A Data Grid Table
 - A Data Grid Form
 - Can You Show Me Some Examples?
 - Data Grid Table
 - Data Grid Forms
 - How Do I Create My First Data Grid Table?
 - Locate Data Grid on Tools Palette
 - Drag Data Grid Onto Card
 - Populate Data Grid Using The Property Inspector
 - Customizing Columns
 - Populate Data Grid Using dgText Property
 - Populating the Data Grid Using dgData Property

- How Do I Create My First Data Grid Form?
 - Locate Data Grid on Tools Palette
 - Drag Data Grid Onto Card
 - Change Style to "Form"
 - Add Some Text
 - Result
- Example: Creating a List of People
 - Attached Files
 - What You Will Create
 - Add Data Grid to Card
 - Change Data Grid Style
 - The Data That Is Going to Be Displayed in Each Row
 - Edit the Row Template
 - Edit the Row Template Group
 - Rename Label Field
 - Make Name Field Bold
 - Add Title Field
 - Change Text Alignment
 - Add an Image and Resize the Background Graphic
 - Stop Editing the Row Template Group
 - Edit the Row Behavior
 - The FillInData Message
 - The LayoutControl Message
 - Set the Data of the Control
 - The Result
 - Troubleshooting: Row Height
- Data Grid Fundamentals
 - What Is This "Data Grid Templates" Stack That Appeared in My Project?
 - Data Grid Templates Stack
 - Data Grid Templates Card Contents
 - Opening the Data Grid Templates Stack
 - What Is a Row Template?
 - Row Template
 - How a Record Template Is Used
 - What Is a Column Template?
 - How a Column Template Is Used
 - How Do I Populate a Data Grid With Data?
 - Use The Property Inspector
 - Set the dgText Property
 - Set the dgData Property
 - How Do I Customize a Form's Row Template?
 - Reveal the Row Template
 - Example of a Customized Template
 - Edit Behavior

- Edit the Behavior Script
 - An Example of a Customized Behavior
- How Do I Customize a Table's Columns?
 - Use Property Inspector to Create a Column Template
 - Edit the Row Template Group
 - Edit Column Group
 - A Column Template
 - Supporting Controls
 - Behavior Example
 - Result
- Working With Data Grids (Forms & Tables)
 - How Do I Determine the Selected Line?
 - The dgHilitedLines Property
 - Determining the Line in a Row's Behavior
 - How Do I Get Data Associated With a Row or Column?
 - Getting Data Associated With the Selected Row
 - Getting Data When the Selection Changes
 - Getting Data in a Row Behavior
 - Getting Data in a Column Behavior
 - How Do I Add a Row of Data to an Existing Data Grid?
 - Using AddData
 - Using AddLine
 - Scrolling Data Into View and Getting the Data Control
 - How Do I Update Data in a Row?
 - Updating a Row's Data and Refresh Data Grid Automatically
 - Updating a Row's Data Without Refreshing the Data Grid
 - How Do I Clear Data From a Data Grid?
 - How Do I Add a mouseDown Event to the Data Grid Without Breaking It?
 - What Doesn't Work
 - What Does Work: dgMouseDown
 - How Can I Store an Option Menu Value When the User Makes a Selection?
 - Example Data Grid
 - Edit Column Behavior
 - FillInData
 - menuPick
 - The Behavior in Action
 - How Do I Refresh a Data Grid After Making Changes to a Template Through Script?
 - How Do I Use a Template in Multiple Data Grids?
 - Create First Data Grid
 - Create Other Data Grids
 - Result

- How Can I See What the Data Grid's Internal Array Currently Looks Like?
 - Using PrintKeys
- How Do I Get Aggregate Values for Columns?
 - Add a getProp to Your Data Grid Script
 - Using the Custom Property
 - The Result
- How Do I Determine if the Data Grid Has Focus?
- How Do I Export Data From a Data Grid?
 - The Example Data Grid
 - The Handler for Populating Data Grid
 - Export Handler
 - Example Output
- How Do I Work With Checkboxes in a Data Grid?
 - Adding a Checkbox to a Form
 - Edit Row Template Group
 - Add a Checkbox
 - Edit Checkbox Properties
 - Delete the Label Field
 - Edit Row Behavior
 - Edit the FillInData Handler
 - Update LayoutControl and ResetData
 - Update Data Grid Row Value When User Changes Checkbox State
 - Populate the Data Grid
 - Test
 - Checking/Unchecking All Checkboxes in a Data Grid
 - Test
- Working With Data Grid Tables
 - How Do I Change Column Alignment?
 - Using the Property Inspector
 - Using Script
 - How Do I Sort by a Column?
 - Using the Property Inspector
 - Using Script
 - How Do I Resize Columns?
 - Using the Property Inspector
 - Using Script
 - How Do I Override the Default Behavior for Rendering Data to a Cell?
 - Begin With a Data Grid Table
 - Create a Button
 - Set the Script of the Button
 - Customize Behavior
 - Set 'Default Column Behavior' Property

- Example: Truncate Tail
 - Refresh
 - Example: Coloring Cells
 - Result
- How Do I Determine if a User Clicks in the Table Header?
 - The Table Header
 - Example
- How Do I Display a Contextual Menu When the User Clicks on a Column Header?
 - Overview
 - The Code
 - The Result
- What if I Need to Work With `htmlText` or `rtfText`?
 - Option 1: Set the `dgProp["Default Column Behavior"]` Property
 - Option 2: Create a Custom Column Template
- How Do I Display Line Numbers in a Table?
 - The Table
 - Add Line Number Column
 - Reorder and Create Column Template
 - Edit Line Number Group
 - Edit Column Behavior
 - Edit Behavior Script
 - Refresh Data Grid
 - The Result
 - But How Do I Set the `dgText` Then?
- How Do I Customize Column Sorting?
 - A Data Grid Table
 - The Data Grid Group Script
 - The Result
 - Custom Sort Example
- How Do I Disable Column Sorting?
 - Sort By No Column
 - Edit Data Grid Script
 - Add `SortDataGridColumn` Handler
- How Do I Perform an Action After the User Sorts a Data Grid?
 - Add `SortDataGridColumn` To Data Grid Group Script
- How Do I Align Decimals in a Column?
 - Create a Data Grid Table
 - Add Some Data
 - Create Column Template
 - Edit Column Template
 - Edit Column Template Group
 - Create 3 Fields

- Edit Column Behavior
 - Update FillInData
 - Update LayoutControl
 - Update EditValue
 - Refresh Data Grid
 - Test
- How Can I Colorize Individual Lines in a Table?
 - Attached Files
 - Coloring a Single Line
 - The Default Custom Column Behavior
- Working With Data Grid Forms
 - How Do I Create a Form With Variable Line Heights?
 - Turn Off "Empty row height"
 - Edit Row Template
 - Turn Off dontWrap Property
 - Edit Row Behavior
 - Script Field to Resize to Fit Height
 - Refresh Data Grid Contents
 - How Do I Sort Records by a Specific Key's Values?
 - Example
 - Option Menu Code
 - The Result
 - How Do I Create Rows That Can Expand/Contract?
 - Attached Files
 - Expanding and Contracting Rows
 - Setting the Data Grid Properties
 - Updating the Row Behavior Script
 - How Can I Speed Up Drawing When "Fixed Row Height" Is False?
 - CalculateFormattedHeight
- Using the Built-in Field Editor
 - How Do I Open a Table Cell for Editing?
 - What You Need to Know
 - EditFieldText
 - EditValue
 - EditCell and EditCellOfIndex
 - CloseFieldEditor
 - How Can the User Edit Field Content in a Data Grid Form?
 - What You Need to Know
 - EditFieldText
 - EditValue
 - EditKey and EditKeyOfIndex
 - CloseFieldEditor

- How Can I Edit the Text as UTF-8, UTF-16 or HTML?
 - The Default Behavior
 - Changing the Default Value For the Editor
 - The Result
- How Can I Select the Text in the Edit Field When It Opens?
 - The Default Behavior
 - Selecting The Text
 - An Example Script
- How Do I Save Changes the User Makes in an Editor Field to an External Data Source?
 - When Calling `EditFieldText` With 3 Parameters (Simpler)
 - When Calling `EditFieldText` With 1 Parameter (More Flexible)
- How Can I Customize the Field Editor Behavior?
 - Create Your Behavior Script
 - Customize Your Script
 - Assign Your Custom Behavior to Field Editor
- Building Standalones With the Data Grid
 - What Do I Need to Do to Deploy a Standalone With a Data Grid?
 - How The Standalone Builder Adds The Necessary Files
 - But What About Launcher Stacks (Splash Stacks)?
- Useful Things to Know
 - What Sorts of Things Should I Not Do in Order to Avoid Needless Suffering?
 - Don't Call a Handler That Redraws the Data Grid From Within a Control in the Data Grid
 - Don't try to draw a Data Grid on a card that is not open
 - Do not lock messages when accessing data grid properties.
 - Do not password protect the Data Grid Templates Stack
 - Don't Rename the "Data Grid Templates" Stack
 - Don't Try to Search When Data is Being Loaded From an External Source
 - Stop Editing the Template Group Before Drawing Your Data Grid
- Advanced Options
 - Displaying Large Amounts of Data
 - Attached Files
 - The `dgNumberOfRecords` Property
 - `GetDataForLine` Callback
 - Creating a Data Grid by Hand
 - Copy Data Grid From `revDataGridLibrary`
 - Set The "style" Property
 - Assign a Row Template
 - How Do I Create a Datagrid Dynamically From Tab Delimited Data

- Prepare Your Data
 - Create a data grid
 - Create the Row template
 - Fill in Your Data
 - Prepare the Columns of the Data Grid
 - Everything together
- Data Grid Tips & Tricks
 - How Do I Scroll a Row to the Top of the Data Grid Form?
 - How To Do It
 - The Result
 - How Do I Find Out Which Line the Mouse Is Over?
 - Converting a Database Cursor to a Data Grid Array
 - The Handler
 - How to use
 - Example
- Printing In LiveCode
 - Introduction
 - Controlling Printer Device Settings
 - Choosing a Printer
 - Choosing Output Mode (e.g. Print to File)
 - Working with Printer Dialogs
 - Saving Printer Settings
 - Paper Related Options
 - Job Related Options
 - Printer Font Metrics (Windows)
 - Printing a Card
 - Card Layout Options
 - Printing Fields & Text
 - Printing a Layout
 - Printing a complex layout
 - Printing Multiple Pages
 - Multiple pages using card printing
 - Multiple pages when using layout printing
 - Working with Scrolling Fields when Layout Printing
 - Working with Print Ranges
 - Printing a Browser Object
- Processing Text and Data
 - Introduction
 - Using Chunk Expressions

- Types of Chunks
- Using Chunks with Containers
- Using Chunks with Properties
- The basic chunk types
 - The Character Chunk
 - The Segment Chunk
 - The Item Chunk and the itemDelimiter Property
 - The Line Chunk and the lineDelimiter Property
- Other chunks
- Specifying a Chunk
- Negative Indexes in Chunk Expressions
- Complex Chunk Expressions
- Using Parentheses in Chunk Expressions
- Nonexistent Chunks
 - Nonexistent character or segment:
 - Nonexistent item:
 - Nonexistent line:
- Specifying a Range
- Counting the Number of segments, Lines or Items
- Comparing and Searching
 - Checking if a Part is within a Whole
 - Case Sensitivity
 - Checking if text is True, False, a Number, an Integer, a Point, a Rectangle, a Date or a Color
 - Check if a segment, Item or Line Matches Exactly
 - Check if one String Starts or Ends With Another
 - Replacing Text
 - Retrieving the Position of a Matching Chunk
 - Chunks Summary
- Regular Expressions
 - Searching using a Regular Expression
 - Replacing using a Regular Expression
 - Filtering using a Wildcard Expression
- International Text Support
 - What are Text Encodings?
 - What are scripts?
 - Introduction to Unicode
 - Creating Unicode Apps
 - Things to look out for
- Using Arrays
 - When to Use Arrays
 - Array Functions in LiveCode
- Encoding and Decoding
 - Styled Text

- URLs
 - Binary Data – Base64 (for MIME Email Attachments and Http Transfers)
 - Binary Data – Arbitrary Types
 - Character to Number conversion
 - Data Compression
 - Encryption
 - Generating a Checksum
- XML
 - The XML Tree Structure
 - When to use XML
 - Methods for Handling XML in LiveCode
 - The XML Library: Loading, Displaying and Unloading XML
 - The XML Library: Editing XML
- Sorting
 - The Sort Container Command: Overview
 - The Sort Container Command: Using Sort Keys
 - The Sort Container Command: Sorting Randomly
 - The Sort Container Command: Stable Sorts – Sorting on Multiple Keys
 - Sorting Cards
- Transferring Information with Files, the Internet and Sockets
 - Introduction
 - File Name Specifications and File Paths
 - What is a File Path?
 - Locating a file
 - The structure of a file path
 - File paths on Windows systems
 - File paths on OS X systems
 - Folder paths
 - File paths for OS X bundles
 - Moving, renaming, or deleting a bundle
 - Referring to files inside a bundle
 - The / character in a file or folder name
 - Absolute and Relative File Paths
 - Absolute file paths
 - Relative file paths
 - Finding the current folder
 - Going up to the parent folder
 - Going up multiple levels
 - Starting at the home directory
 - When to use relative and absolute file paths
 - Special Folders
 - File Types, Application Signatures & File Ownership
 - Windows File Extensions and Ownership
 - Applications that don't own files

- Applications that own their own files
 - Installing custom icons
 - Custom application icons
 - Custom file icons
 - File extensions
- OS X File Types and Creators
 - Applications that don't own files
 - Applications that own their own files
 - Creating Files
 - File extensions
- Mac OS X Classic File Types and Creators
 - Applications that don't own files
 - Applications that own their own files
 - Installing custom icons
 - Custom application icons
- Linux File Extensions
- Working with URLs
 - An Overview of URLs
 - URL Schemes
 - The http scheme
 - The ftp scheme
 - Directories on an FTP server
 - The file scheme
 - File path syntax and the file scheme:
 - Conversion of end-of-line markers
 - The binfile scheme
 - The resfile scheme
 - Creating a resource fork
 - Manipulating URL contents
 - - The URL keyword
 - Using the content of a URL
 - Putting data into a URL
 - Chunk expressions and URLs
 - Putting data into a chunk
 - **URLs and memory**
 - **Deleting URLs**
- Uploading and Downloading Files
 - Uploading using the put command
 - Downloading using a URL
 - Non-blocking transfers

- Using the load command
 - Checking status when using the load command
 - Canceling a file transfer & emptying the cache
 - Uploading and downloading large files using FTP
 - Using callback messages
 - Uploading, downloading, and memory
- Using a stack on a server
 - Using a compressed stack
 - Saving stacks from a server
- Other Internet Commands
 - Launching the User's Browser with a URL
 - Working with Web Forms
 - Working with FTP
 - HTTP methods and http URLs
 - HTTP headers
 - Additional Transmission Settings
 - Troubleshooting
- revBrowser – Rendering a Web Page within a Stack
- SSL and Encryption
 - Encrypting and Decrypting Data
 - Connecting using HTTPS
 - Implementing your own secure protocols
- Writing your own protocol with sockets
 - Opening a connection
 - Looking up a host name or IP address
 - Reading and writing data
 - Disconnecting
 - Listening for and accepting incoming connections
 - Handling errors
- Working With Databases
 - Introduction
 - Introduction to Database Access
 - SQL Databases
 - The Basics of Database Structure
 - SQL and Record Sets – Database Cursors
 - Choosing a Database
 - Reasons to Choose a Database Type
 - Overview of ODBC
 - Performance for Direct Access Versus ODBC Access
 - Software for Database Access
 - Finding Database Drivers

- MySQL
 - PostgreSQL
 - SQLite
 - ODBC managers and database drivers
- Working with Media
 - Introduction
 - Bitmap Images
 - Importing Images
 - Importing Referenced Images
 - Import using Screen Capture
 - Creating Images
 - Using the Paint Tools
 - Scripting with the Paint Tools
 - Manipulating Binary Image Data
 - Rendering an Image from Objects
 - Exporting Images
 - Copying and Pasting Images
 - Working with Animated GIFs
 - Working with Vector Graphics
 - The Vector Graphic Tools
 - Creating Graphics by Script
 - Manipulating graphics by script
 - Working with Video
 - The Player Object
 - Working with Sounds
 - Importing an Audio Clip
 - Playing an Audio Clip
 - Working with Visual Transition Effects
 - Creating Custom Skins
 - Custom Themed Buttons
 - Irregular Windows
 - Blend Modes (transfer modes or inks)
 - Full screen mode
 - Displaying a Backdrop
- The LiveCode IDE
 - Introduction
 - The Menu Bar
 - The File Menu
 - The Edit Menu
 - The Tools Menu
 - The Object Menu
 - The Text Menu
 - The Development Menu
 - The View Menu

- The Window menu
 - The Help Menu
- The Property Inspector
 - Property Inspector settings
- The Script Editor
 - Main Script Area
 - Breakpoints Area
 - Apply Button
 - Errors Pane
 - Handler List
 - Documentation Pane
 - Search Results Pane
 - Script Tabs
- The Debugger
 - Continue
 - Stop Debugging
 - Show Next Statement
 - Step Into Next Statement
 - Step Over Next Statement
 - Step Out
 - Debug Context
 - Variable Watcher
 - Breakpoint Management
- The Script Editor Menubar
 - The File Menu
 - The Edit Menu
 - The Debug Menu
 - The Handler Menu
 - The Window Menu
 - Autocomplete
 - Using Autocomplete
 - Autocomplete Pro
 - Autocomplete Snippet Manager
- The Message Box
 - Message Box settings
 - Single and Multiple Line Modes
 - Global Properties
 - Global Variables
 - Pending Messages
 - Front Scripts & Back Scripts
 - Stacks In Use
- The Toolbar
- Find and Replace
- Keyboard Shortcuts
- Deploying Your Application

- Introduction
- Building a Standalone Application
 - Step by step
- Standalone Applications Settings
- General Settings
 - Including Additional Stacks
 - Including Additional Files
 - Including Additional Resources
- Mac Deployment Settings
- Windows Deployment Settings
- Linux Deployment Settings
- iOS Deployment Settings
- Android Deployment Settings
- HTML5 Deployment Settings
 - Bug Reports
- Testing Your Application
- Extending LiveCode
 - Introduction
 - LiveCode Builder Extensions
 - Creating LiveCode Builder Extensions
 - Extension Builder
 - Create your own simple widget
 - Create a .lcb file
 - Declare Type and Identifier
 - Declare Metadata
 - Importing libraries
 - Core Handlers
 - Draw a Pink Circle
 - Test the Code
 - Properties
 - Loading and Saving widget data
 - Understanding Error Messages
 - LiveCode Builder API
 - Hello World library
 - Loading and Creating Widgets Manually
 - Extensions Course
 - Canvas API
 - Using path objects
 - Rendering text
 - Transformations
 - Composed Widgets
 - A simple composed widget

- Using native APIs for libraries
 - C
 - Java
 - Other binding examples
 - Objective-C
 - Using native views for widgets
 - Android
 - iOS
 - Mac
 - Windows
 - Linux
 - HTML5
- The extension package
 - Documenting Your Extension
 - Adding API Markup
 - Adding a User Guide
 - Packaging Extensions
 - Package requirements
- The Extension Store
 - Registering as an Extension Developer
 - Uploading Extensions to the Store
 - Activating your Extension
 - The Extension Store in the Extension Manager
- Other ways to extend the Built-in Capabilities
 - Communicating with other process and applications
 - Reading and writing to the command shell
 - Launching other applications
 - Closing another application
 - Communicating with other processes
 - Using AppleScript and VBScript (Open Scripting Architecture or AppleEvents
 - Using Local sockets
- Externals – code written in lower level languages
 - The Externals SDK
- LiveCode Builder Bytecode Reference
 - Introduction
 - Operations
 - Jump
 - Jump If False

- Jump If True
 - Assign Constant
 - Assign
 - Return
 - Invoke
 - Invoke Indirect
 - Fetch
 - Store
 - Assign List
 - Assign Array
 - Reset
- LiveCode Builder Language Reference
 - Introduction
 - Tokens
 - Comments
 - Line continuation
 - Case-Sensitivity
 - Typing
 - Modules
 - Metadata
 - Imports
 - Definitions
 - Constants
 - Types
 - Handler Types
 - Record Types
 - Variables
 - Handlers
 - Foreign Handlers
 - The C binding string
 - The Obj-C binding string
 - The Java binding string
 - Properties
 - Events
 - Statements
 - Variable Statements
 - If Statements
 - Repeat Statements
 - Throw Statements
 - Return Statements
 - Transfer Statements
 - Call Statements
 - Bytecode Statements
 - Unsafe Statements
 - Expressions
 - Constant Value Expressions

- Variable Expressions
 - Result Expressions
 - List Expressions
 - Array Expressions
 - Call Expressions
- Namespaces
- Experimental Features
 - Safe Foreign Handlers
 - Foreign Aggregate Types
- LiveCode Builder Style Guide
 - Introduction
 - Copyright headers
 - Naming
 - Module name
 - Naming variables and parameters
 - Naming handlers
 - Naming types
 - Documenting the source code
 - Named constants
 - Whitespace
 - Indentation
 - Wrapping
 - Handler declarations, definitions and calls
 - List and array literals
 - Widget-specific guidelines
 - Writing load handlers
 - Classic control properties
 - Host control properties
 - Events
- LiveCode Documentation Format Reference
 - Introduction
 - Format
 - Element names
 - Name (required)
 - Synonyms
 - Type (required)
 - Syntax (required for API-level entries)
 - Summary (required)
 - Associated
 - Introduced
 - OS
 - Platforms
 - Example
 - Parameters
 - Value and return parameters

- [Description \(required\)](#)
 - [References](#)
 - [Tags](#)
- [Other documentation syntax](#)
- [Separate docs files](#)
- [Inline documentation](#)
- [Separate docs files examples](#)
 - [Dictionary example](#)
 - [Library example](#)
- [Inline examples](#)
 - [LiveCode Builder syntax example](#)
 - [LiveCode Builder handler example](#)
 - [LiveCode script handler example](#)
- [Overview](#)
- [Known issues](#)
- [Breaking changes](#)
 - [Boolean constants](#)
 - [Infinity constant](#)
 - [Implicit object](#)
- [Platform support](#)
 - [Windows](#)
 - [Linux](#)
 - [Mac](#)
 - [iOS](#)
 - [Android](#)
 - [HTML5](#)
- [Setup](#)
 - [Installation](#)
 - [Uninstallation](#)
 - [Reporting installer issues](#)
 - [Activating LiveCode](#)
 - [Command-line installation](#)
 - [Command-line uninstallation](#)
 - [Command-line activation](#)
- [Previous release notes](#)

LiveCode Script

Introduction

Before getting started with LiveCode, there are a number of things you may want to know. This guide introduces LiveCode, tells you where to begin, how to install, register and how to use the documentation.

Welcome

LiveCode makes it easy to turn your concepts, ideas or designs into powerful applications. You can create desktop, Internet, database and network applications. LiveCode includes a complete graphical user interface builder, as well as the tools you need to produce faceless or background applications.

LiveCode is easy to learn. If you are a complete beginner to the world of programming, you will find you can become productive quickly. If you are an experienced programmer, you will find the environment extremely powerful and one of the most productive you have used.

LiveCode's programming language is object-based. This makes it easy to write modules of code that are attached directly to individual objects. Unlike other languages, LiveCode lets you both run and edit an application live. Traditionally the run, edit, compile and debug cycle in programming takes up a lot of time. In LiveCode all of these activities can take place together.

If you're coming from another language, you'll appreciate that the LiveCode language is typeless, with data automatically stored in the most efficient format and converted, so you don't have to learn how to deal with different types of data. Memory management is fully automatic. LiveCode applications are not interpreted in the traditional sense, so they provide excellent performance. A LiveCode graphical application will often feel more responsive to use than a Java application, and take less time to write.

A LiveCode application is completely cross-platform. That means you can run any application you create on all major modern operating systems, including Windows, Linux, Unix and Mac OS. Unlike most other cross-platform frameworks, a LiveCode application will always look and feel native on each platform that you deliver it on. You can take advantage of many platform specific features on each platform.

LiveCode removes the need to learn how to access most individual programming interfaces for each operating system. This saves you the time and effort of learning thousands of platform-specific interface calls.

The goal of this manual is to provide the most accessible, comprehensive and useful guide to LiveCode possible. We aim to provide a level of depth that covers the advanced feature set, while remaining accessible to beginners. We hope you find this material useful and enjoy using LiveCode.

Where to Begin

We recommend you start by taking a look at [our online resources](#). There is a wide variety of introductory material there, including step-by-step examples and guides for performing common tasks.

System Requirements

Memory and disk requirements below are for the *LiveCode development environment*, not for applications created by LiveCode. Applications created with LiveCode will require sufficient free system resources to load, display, process and interact with the content of the application. Because this will vary considerably depending on the application created, it is not possible to publish a list of requirements for

applications built with LiveCode. You will need to test your application to determine the minimum requirements. Many LiveCode applications will run on a moderately powerful computer and require substantially fewer resources than those listed for the development environment, others will require more.

All Operating Systems

To use LiveCode you will need:

- 1024x768 or larger monitor
- True color display (16-bit or 32-bit depth)
- At least 256Mb of memory
- At least 150Mb of disk space

Requirements for Windows Systems

LiveCode supports the following versions of Windows:

- Windows 2000 SP4
- Windows XP SP2 and above
- Windows Server 2003
- Windows Vista SP1 and above (both 32-bit and 64-bit)
- Windows 7 (both 32-bit and 64-bit)
- Windows Server 2008

Requirements for Linux Systems

The minimal requirements for LiveCode to run on Linux are:

- 32-bit installation, or a 64-bit linux distribution that has a 32-bit compatibility layer
- 2.4.x or later kernel
- glibc 2.3.2 or later X11R5 capable Xserver running locally on a 24-bit display*
- compositing window manager (optional - required for alpha-blended window shapes)
- gtk/gdk/glib (optional - required for native theme support)
- pango/xft (optional - required for pdf printing, anti-aliased text and unicode font support)
- lcms (optional - required for color profile support in JPEGs and PNGs)
- gksu (optional - required for elevate process support)
- mplayer (optional - required for video playback)
- esd (optional - required for audio playback)

Although impossible to test every existing Linux distribution, we are aiming to ensure that LiveCode runs on as wide a variety of systems as possible. To achieve this, the engine has been implemented to have minimal direct dependencies on system software, and will gracefully degrade in feature set if it cannot find the libraries it needs. Generally any recent linux distribution including Gnome/GTK support will have the required libraries for full feature support – for example, Ubuntu 7 supports all these features (although alpha blended window shape support requires you to be running with 'Advance Desktop Effects' turned on).

Requirements for Mac OS X Systems

LiveCode supports the following versions of Mac OS X:

- 10.6.x (Snow Leopard)
- 10.7.x (Lion)
- 10.8.x (Mountain Lion)
- 10.9.x (Mavericks)
- 10.10.x (Yosemite)
- 10.11.x (El Capitan)

Using the Documentation

Documentation Conventions

Language Terms

This documentation contains examples of LiveCode language syntax. The syntax for each language term is provided in **boldface text**.

The syntax description uses the standard conventions to indicate parts of the syntax:

- `[]` Square brackets enclose optional portions.
- `{ }` Curly braces enclose sets of alternatives from which to choose.
- `|` Vertical bars separate different alternatives.
- `\` Line continuation character – this line continues to the next line.
- *Italics* indicate placeholders that you don't enter literally.

Code examples are provided in `typewriter text` or

`code blocks`

Menu Reference

When you are told to choose an item from a menu, the menu name is written, followed by an arrow, followed by the menu item. For example **File -> New** means choose "New" from the "File" menu.

Keyboard Shortcuts

Many menu items have keyboard equivalents. These items can be accessed by holding down a modifier key and pressing another key. The modifier keys used throughout this document are generally the modifier keys used on Windows, Linux and Unix. If you are using Mac OS, you may need to substitute modifier keys using the table below. For example, if you are using a Mac, where we say press **control-x** in this documentation, you may need to press **command-x**.

Windows and Linux keyboard shortcuts	Mac equivalent
Control	Command
Alt	Option

Figure 2- Cross-platform Keyboard Shortcuts

Screen Snapshots

Screenshots are used extensively throughout the documentation. Please remember that LiveCode is a

cross-platform application. While many of the snapshots are taken on the Windows platform, most of the feature set will work on Linux, Unix and Mac OS. Snapshots of these platforms are used from time to time and are also used to illustrate platform specific features.

Note Styles

Tip: This style of note tells you about an alternative way to do something or describes a shortcut.

Important: This style of note describes an aspect of LiveCode that may confuse you if you don't know about it and tells you how to avoid unexpected behavior.

Note: This style of note provides some additional information that may be useful, or emphasizes a key point.

Caution: This style of note warns you about a potential problem that might cause an error or loss of data.

Navigating the Documentation

The LiveCode documentation is spread across four areas; the Start Center, the Dictionary (API), the Guides and other resources.

The Start Center is aimed at introducing you to LiveCode, and should be your first port of call if you have not used LiveCode before. The dictionary, or API, is a reference resource which you can refer to while implementing your projects. Guides are online resources at livecode.com. Other resources include Sample Stacks, a free, online repository of user-contributed stacks, User forums, and the resources page at livecode.com/resources/.

You can access the documentation from within LiveCode from the **Help** menu. Additional resources are available at www.livecode.com.

Start Center

The purpose of the Start Center is to introduce you to LiveCode and provide you with enough knowledge to start implementing your own projects. It can be accessed by choosing **Help -> Start Center**.

The first time you open the Start Center you will see an invitation to take the Interactive Tour, a step-by-step tutorial that guides you through creation of a simple project in LiveCode. If you choose to skip the tour you may always enter it later by clicking the Interactive Welcome button in the Start Center.

The Start Center itself is split across three sections—quick links to the LiveCode pages on various social media sites; a collection of quick link buttons, which take you to various helps and guides; and a list of your most recently opened stacks.

Social Media Links

LiveCode maintains pages on popular social media sites, such as Facebook, YouTube, LinkedIn and Stack Overflow. These can help you keep up to date on LiveCode product announcements, professional networking, and help and training resources.

Recent Stacks

This shows a list of your most recently opened stacks. Double click on a stack to open it and close the Start Center window.

Quick Links

The quick link buttons let you quickly access tutorials, sample projects, guides, LiveCode blog posts, and online user forums. Here is an overview of these links:

Interactive Welcome

Enter the Interactive Tour, a step-by-step tutorial that guides you through creation of a simple project in LiveCode.

New

Click this button to create a new stack. You will see a menu of common stack types and sizes to choose from. Once you choose your stack size or type the new stack will be created and the Start Center menu will close.

Sample Projects

The Sample Projects button opens an online repository of sample stacks submitted by LiveCode developers. The repository contains dozens of stacks that you can download for free and use as you wish.

Tutorials

Opens the LiveCode online tutorial web page. From here you can choose one of dozens of lessons and how-to tutorials that give you step-by-step instructions on how to create things, from a simple "hello world" application to simple games to sophisticated graphical effects.

In Depth Guides

Opens the LiveCode Dictionary. Here you can choose to enter the API documentation, the definitive reference of all of the terms in the LiveCode scripting language. Or you can open one of several user guides, which describe in detail how to develop and deploy apps in LiveCode.

Forums

LiveCode has a very active and friendly user community. We encourage all new users to sign up to our mailing list and user forums as they are a great way to get technical help, meet new people and keep up-to-date with what is going on in the world of LiveCode. The "User Forums" section provides a link to our forum registration page as well as a list of links to the most recent posts which have been made.

Blog

This button takes you to LiveCode's online blog, where LiveCode engineers and developers regularly post updates about what's new in LiveCode, how-to tutorials, and announcements of interest to the LiveCode developer community.

Account

Click this button to log in to your LiveCode online store account. There you can see your purchases, as well as look for useful add-ons to make LiveCode and even more powerful development tool.

Dictionary

The LiveCode Dictionary contains detailed reference materials for developing applications with LiveCode. It is divided into two main sections—the LiveCode API reference and the LiveCode Guides. It can be accessed by choosing **Dictionary (API)** from the **Help** menu, or by clicking the Dictionary icon on the LiveCode Toolbar.

LiveCode APIs

The LiveCode Application Programming Interface, or API, reference contains the complete syntax for all LiveCode components, including LiveCode Script, LiveCode Builder, and all installed widgets and libraries. If you are new to LiveCode you will likely spend most of your time in LiveCode Script, the scripting language used in developing LiveCode applications. The examples that follow will be taken from the LiveCode Script API, but the API documentation for all components uses the same format and layout.

To search one of the API lists, click on the API tab at top left, then choose the API you want to search from the option list. For example, to search for a LiveCode Script term, choose LiveCode Script.

Searching. If you know what term you want to look up, or if you have a good idea of what it might be, just start typing it into the search box, and a list of terms that match your search term will appear in the entries list.

Filtering. If, on the other hand, you don't know the name of the term, but have a general idea of what terms you want to look at you can filter the list of terms by clicking on a filter topic. For instance, to see a list of messages that buttons can respond to, click "message" under the **type** category and "button" under **associations**. You can clear a filter by clicking again on the selected filter.

Note that the entries list displays only the entry name, token type, and syntax for the entry. You can view the complete entry by clicking on the desired line in the list. This will show the entry in the area underneath the entries list. Each LiveCode scripting term is thoroughly documented with the proper syntax, detailed examples, a detailed description, related terms, and more.

LiveCode User Guides

The User Guides are a complete reference to LiveCode. They detail what the dialogs and windows within the development environment do, and explain the LiveCode language in depth. To open a guide, click on the Guide tab in the Dictionary and choose the desired guide from the list.

RevOnline

RevOnline gives you the opportunity to upload and share your projects as well as download those created

by other members of the LiveCode online community. It can be accessed via the Sample Stacks button located in the menu bar. The main interface can be seen below:

The interface is designed to organise the online content into various categories to aid you in finding the kind of content you wish to download. You will find three main types of content in RevOnline, stacks (LiveCode programs), externals and code snippets. Externals are libraries that can extend the functionality of the LiveCode language using a lower level language. A code snippet is a portion of LiveCode script that you can paste into and incorporate into your own projects.

The results of searching or browsing through the RevOnline content are displayed in the *search list* area. This is an explanation of the controls you can use to adjust the parameters of your search:

Search field	Enter text and press return or click “Go” to search. Matches will be made against the titles of uploaded content, the author’s name and content descriptions.
Type list	Highlight one of the items in this list filter search to only show results of that type. Selecting “All” will display content of all three types.
Tag list	Tag your own and others people’s content with a word or phrase. All tags are listed here alphabetically. Selecting a tag will filter the search results for uploads that have been tagged accordingly. Selecting “All” will remove the filtering.
Search order	Change the order in which the search results are displayed. There are four options. “A to Z” lists content alphabetically by title, “Popularity” by the number of times it has been downloaded, “Rating” by the average user rating (out of five stars) and “Date” by the date on which the content was uploaded to RevOnline
Order direction	The direction (ascending/descending) of the search order can be changed by clicking on the arrow to the right of the sort control.
View mode	The search list area can be viewed in two modes, a vertical scrolling list or a grid. In “list” mode 10 results will be displayed per page, in “grid” mode the number of results displayed will be dependent upon the size of the RevOnline stack. Whilst in “grid” mode the <i>viewing pane</i> area will not be visible.
User filter	If you apply user filter to a search, the user’s name will appear above the type and tag lists (see below), subsequently only content by that user will be returned in the search results. Clicking on the button or text of the user filter will remove it.

The *search list* area displays a preview of the results your search has returned. At the top of this list you will see the *page controls* which allow you to navigate forwards and backwards through the pages of your search should it return more results than can be displayed on one page.

An example of the search list previews can be seen below:

At the top of the preview is the content’s name. You can also see its average user rating out of five stars, the total number of times it has been downloaded, the authors name and a list of tags applied to this upload. The “download” button at the bottom will download and launch this particular stack, it also tells you the size of the download. The *preview image* can either be a custom screenshot as above or one of

three standard images. These are black for stacks, red for code snippets and green for externals. The *type icon* in the lower left corner of the preview image also indicates the type, these are shown below for a stack, code snippet and external respectively:



Clicking on the *preview image* or content title will update the right hand *viewing pane* with more detailed information about this upload. Clicking on the authors name will display the author's user profile in the *viewing pane* and introduce a *user filter* to the search results. When the *search list* is in "grid" mode the *viewing pane* will be hidden, doing either of the above operations will cause the *search list* to switch back to "list" mode and display the associated content in the *viewing pane*.

The *viewing pane* displays detailed information on users and uploads.



If you are a viewing stack details as shown above, in addition to the information provided by the preview you will also see a headline, description and a revision number which is incremented whenever the author uploads a new version of their stack, code snippet or external. Clicking on the author's name or *user profile image* will display their profile view. Clicking on the *content type* or any of the tags in the *list of applied tags* will apply these filters to the search controls. When viewing an external you will see an additional "Platform:" field that indicates which platforms the external library is compatible with, make sure your operating system is supported before downloading and installing an external. If you are currently logged in to your Rev Online account you will be able to give the upload a rating by hovering over the *rating control* and clicking when the rating you wish to submit is displayed, this can only be done once per upload. You can also tag the upload by clicking on the *add tag control* which brings up following dialog:



As you type into the text entry field, tag suggestions will appear in a drop down menu that can be selected using the arrow keys. Once you have finished typing press the ok button to submit your tag, you must be logged in to complete this operation.

The *status bar* at the bottom of the RevOnline interface displays the number of results your search has returned as well as a progress bar indicating the progression of searches and downloads. You will also see text feedback in response to the operations you perform such as "Retrieving stack details ..." and "Content tagged" etc.

In the top right corner of RevOnline you will find the *login controls* which will display your current login status as shown below:



In order to upload, rate and tag content in RevOnline you must have an account, this account is also required for making comments in the wiki docs detailed elsewhere. To register for an account just click on the "Sign Up" control and enter your email in the dialog that appears. Your email address will be used as

your username for logging in, a password will be emailed to you upon signing up. After receiving this email you must hit the activation link in the email before your account can be accessed. Once activated, you can login to your account using the “Log In” control pictured above which will bring up the following dialog:

The “Remember me” check box will cause RevOnline to maintain your username in the “Email :” field between login attempts. The “Sign in automatically” check box will mean that you are automatically signed in upon opening RevOnline, however if you explicitly sign out you will have to re-enter your password regardless of this preference. If you forget your password at any time you can use the “Forgot your password?” link in the login dialog to have a new one emailed to you.

Upon logging in and viewing your profile for the first time you will be presented with your profile view in edit mode where you have the opportunity to enter information about into the highlighted fields and change your profile picture using the *change profile picture* control as shown below.

Clicking on the *change profile picture* control will bring up the following dialog:

Use the *Position Controls* to move the centre of the image up, down, left or right. You can zoom in or out using the *Zoom controls*. Pressing the *Re-centre image* button will reset the image to its original position and zoom level. Once you are happy with orientation of your picture click ok, or cancel to discard your changes. This dialog is also used to change stack preview pictures.

When you have finished editing your profile you can hit the “save changes” button to submit your information, a minimum of a first and last name must be entered before you can save. You can also choose to display your email or not and change your password to one of your choosing using the “Change password” control. You can reach this profile at any time by clicking on the “My Account” control in the *login controls* which will both display your profile and display all your uploads in the *search list*. To exit this view simply remove the user filter or hit the “My Account” control again.

To upload content to RevOnline simply click on the “upload content” in the *login controls*, you will then be presented with the content browser view.

Clicking on the tabs at the top of this view will change the content type. Above you can see the code snippet view, it simply requires you to paste a section of code into the *code entry field* and press ok, if you have your code snippet stored in your clipboard you can simply press the “paste” button. For stacks you can either select a stack from the list of recently opened stacks or browse to the files location using the “browse” button. For externals you need to check which platforms your external will support and then browse to the appropriate file you wish to upload (shown below). The default file types for extensions on each platform are; “.dll” for Windows , “.bundle” for Mac OS X and “.so” for Linux.

When uploading a code snippet or external RevOnline will place the code or file into an installer stack. The installer stacks give you the option to “install” or “try” an external file or “copy to clipboard” a code snippet. When using the try option with an external the file will be removed from your installation when

you next exit LiveCode.

Once you have pressed ok you will be presented with the edit mode view for the file you are uploading. Here you can enter a name description and change the preview picture for your upload (note – you cannot edit the title of an external upload it must be the name of one of the files you have included). If you attempt to upload a file that has already been uploaded to RevOnline, you will be taken to the existing version of that file to edit, this avoids duplicate uploads.

Whenever you are viewing content that you have uploaded or your own profile whilst logged in the edit controls will appear above the *viewing pane*, shown below:



By clicking on the “edit profile” or “edit stack” (for stacks etc.) button you are entering edit mode which allows you to make changes to the information shown. When editing your profile you can simply save or discard the changes you have made. When editing an upload you have the additional options of deleting and updating the uploaded file. The delete control will permanently remove your upload from RevOnline. The update control allows you to replace the stack file, code snippet or external library with a newer version. Clicking on this control will present you with the content browser view discussed earlier. After you have chosen a replacement file you need to save your changes in order for the new file to be submitted.



The list of tags applied to your own uploads can be altered by using the “Add/Delete tags” control



which will bring up the following dialog:



This works in a similar way to the previously explained tag dialog except that you can view the entire list of tags and edit which ones you wish to be displayed. When hitting return after entering the name for your tag a new *tag item* will appear in the list (also achieved by pressing the *add tag* button). Tags can be removed by clicking on the appropriate *remove tag button*. If your tag list is too large for one page you can browse through the pages of tags using the *page controls* at the bottom of the dialog. Once you have finished editing the list you can submit your changes by clicking the ok button.

Additional options for the RevOnline interface can be set in the RevOnline tab of the preferences stack. As pictured below:



Here you can set the remember password and user name preferences and change your password to one of your choosing. The search results in RevOnline are cached to speed up performance. You can tweak the size and time for which this cache remains valid using the two slider controls. A larger cache will give you faster performance but will take longer to pickup changes made to the online content made by other users. If the text only searches option is checked RevOnline will only retrieve the text data for the search list previews which can also improve performance.

Additional Resources

If you have a question that isn't answered in this documentation, you can find additional information to help you on our web site. You can also connect with the LiveCode developer community, which includes many experienced developers who offer their expertise for the common good to answer questions and help their fellow developers. You'll find example code, information on planned enhancements, and the latest updates. Visit:

<http://www.livecode.com>

Developer Community

There are a number of ways to engage with the LiveCode community:

- The [forums](#) (section 1.6.3 above), are available for getting help from other developers, for discussing feature requests and future directions, and for receiving announcements from LiveCode.
- Ask a question on [stackoverflow](#).
- Talk about LiveCode on [the LiveCode subreddit](#).
- Join the [LiveCode user group](#) on Facebook.
- Use the [LiveCode mailing list](#) is also available.

Tip: Tip: You can use the Google search engine to search the list archives, using [this link](#)

Technical Support

LiveCode offers *up and running* technical support to all users to assist you in getting the development environment working on your system.

Many editions include additional technical support for more *advanced* problems and questions. Where support is not included, a separate technical support contract for advanced support can be purchased.

Contact us [here](#) or at support@livecode.com.

Getting Started

Creating a simple *graphical application* in LiveCode can take just minutes. First you create a user interface, including any windows, palettes, dialogs you require. Then you populate the user interface with controls, like push buttons, check boxes, text fields or menus. Finally, you use LiveCode's English-like programming language to tell your application how to behave.

Prerequisites

Before you get started, you should be familiar with how to use your computer. You should also be familiar with common applications such as a web browser, word processor and email client. If you are using Linux or Unix, you should be familiar with basic use of the command line. If you are not familiar with using a computer, we recommend you spend some time getting acquainted before trying to learn LiveCode.

Event Driven Programming

A LiveCode application is driven by user actions. LiveCode constantly watches the computer for common actions, such as clicking on a button, typing into a field, sending data across a network, or quitting an application.

Whenever an event occurs, LiveCode sends a message. When writing your program, you decide what messages you want your program to respond to. LiveCode automatically sends each message to the

most relevant object. For example, if a user clicks on a button, LiveCode sends a message to the button. You place code within the button that tells it how to respond to being clicked on.

There are a wide range of possible events. When a user clicks on a button, a series of events are sent to the button. For example, when the mouse first moves within the border of the button a **mouseenter** message is sent. Then a series of **mousemove** messages are sent as the mouse moves over the button. When the mouse button is depressed a **mousedown** message is sent. When the mouse is released a **mouseup** message is sent. You don't have to respond to all of these events. You simply place code within an object to handle the events you want your application to respond to.

Events that are not handled by individual objects can be handled in a number of ways at different levels of your application, in libraries, or they can be ignored. The rules that govern what happens to events that are not processed by an object are described in the section *The Message Hierarchy*.

We'll return to Event Driven Programming in more detail in the section *Coding in LiveCode*.

Object-Based Programming

Any *graphical application* you build using LiveCode will be based on objects. With LiveCode you typically create the objects of your application before writing any code. You can start by drawing the buttons, text fields, and other controls that make up your application. LiveCode operates like other layout, drawing or application development environment. You can select controls by clicking them, move them by dragging them around, resize them, and change their 'layer' to move them closer or further from the 'top' of the interface.

Once you have the objects in place, you can proceed to attach code to each object to respond to the events you want. LiveCode includes a complete graphical development environment that makes it easy to create and edit any kind of user interface. It includes objects for all the basic operating system elements, including buttons, checkboxes, text fields, menus, graphics, and many more. In addition you can create and customize your own objects that look and behave however you want.

If you are writing a non-graphical application, you can choose to create objects to assist in organizing your code into sections and load these objects off screen, or you can simply write your code in a text file and run the text file directly. This method is commonly used to communicate with Apache and other web browsers when using LiveCode to build server-side or network applications.

Object-Based programming is covered in more detail in the section *Building a User Interface*. Non-graphical applications are covered in more detail in the section *Building a Web Application*.

The 'Edit' and 'Run' Mode

Important: In order to enter **run** mode, choose the browse tool in the *top left* of the tools palette.

In order to **edit**, choose the pointer tool from the *top right* of the tools palette.

Unlike most other development systems, a LiveCode application can be created, edited, debugged and run live.

When in run mode, objects receive all the normal messages that drive a LiveCode application. For example, clicking on a button in run mode will cause a **mouseup** message to be sent to it, causing the button's script to run if you've designed it to respond to the **mouseup** message.

When in edit mode, objects do not receive messages when you click on them, and you can move, resize or edit the properties for objects.

Tip: To temporarily stop all messages being sent to your application while editing it, choose *Suppress Messages* from the *Development* menu or *Toolbar*.

There are a few other differences between the two tool modes. You can view and edit properties and code in

either mode. Your application does not stop running while you make changes to it. Only mouse interaction with objects is suspended in edit mode to allow you to edit them more easily.

Because LiveCode is constantly live, you can easily make simple changes and watch each change take effect as you make it. This allows you to design and experiment using an iterative process, resulting in a more productive and satisfying development experience.

Structuring your Application

Cards, Stacks & Files

The first step in creating a LiveCode application is creating a window, which in LiveCode is called a stack. Each window you see in LiveCode is a stack. Palettes, dialog boxes, and standard windows are all stacks.

Each stack contains one or more sets of information called cards. Each card can have a different appearance or all the cards in a stack can look the same. By going from card to card in a stack, you change what's being displayed in that stack's window. You can think of a LiveCode stack as a stack of playing cards (hence the name), where you can flip through the cards, but only one card at a time is visible. A stack can have a single card or many cards. See Figure 3 – Stack file Structure, below.

All user interface objects (controls) are created by dragging them on to a card area.

You can also group controls together if you want them to operate as a set. For example, if you have a set of navigation buttons that go from card to card in your stack, you can make them into a single group. Groups can appear on more than one card, so your navigation buttons or background image can appear on each card of your stack. For more details, see the section on *Group and Backgrounds*.

A collection of stacks can be saved together in a single file. This file is known as a stack file. The first stack in the file is called the main stack and will be loaded automatically when your application is run.

The Structure of a Stack File

Each LiveCode file contains one or more stacks: either a single main stack, or a main stack and one or more substacks. Since each stack is a window (including editable windows, modeless and modal dialog boxes, and palettes), a single stack file can contain multiple windows.

You can use this capability to bundle several related stacks into a single file for easy distribution, to organize your stacks into categories, or to allow several stacks to inherit properties from the same main stack.

Figure 3 – Stack File Structure

Opening a Stack File

When you open a stack file, either by using the "Open Stack" menu item in the File menu or by using one of the navigation commands (see **open**, **go**, **modal**, **modeless**, **palette**, or **topLevel** in the *LiveCode Dictionary*), the stack file's main stack opens automatically to its first card.

Important: A stack file is saved as a whole. If you save a stack, all the other stacks in the same stack file are saved at the same time.

Substacks in the stack file do not open automatically when the stack file opens. You must open a substack in a handler or the Message Box, or by using the Application Browser.

Main Stacks and Substacks

The first stack created in a stack file is called the *mainstack*. Any other stacks created in the same stack file are called *substacks* of the mainstack.

The mainstack is part of the object hierarchy of all other stacks in the same stack file. In other words (for the purposes of inherited properties and shared behaviors), the mainstack *contains* its substacks. Events that are not handled by a substack are passed on to the mainstack's script, color and font properties are inherited from the mainstack by its substacks. For more details on the object hierarchy and inheritance in LiveCode in general, see the section on *The Message Path*.

Dialog boxes and palettes are commonly stored as substacks of the main application window, which is typically a mainstack. This allows you to store code and common functions used by all the substacks in the mainstack's script. Because mainstacks are part of the object hierarchy of their substacks, the substacks can call this functionality from scripts within the substack.

Stacks, Stack Files, and Memory

A stack file can be loaded into memory without actually being open. A stack whose window is closed (not just hidden) is not listed in the **openStacks** function. However, it takes up memory, and its objects are accessible to other stacks. (For example, if a closed stack loaded into memory contains a certain image, you can use the image as a button icon in another stack.)

If one stack in a stack file is loaded into memory, so are any other stacks in the same stack file. You cannot load one stack in a stack file without loading all the rest at the same time even if you open only one of the stacks.

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

A piece of code in another stack read or set a property within the closed stack. This automatically loads the referenced stack into memory.

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

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

Tip: To manipulate a stack window in an external, use the **windowID** property. For more information, see the *LiveCode Dictionary*.

Media & Resources

When planning a project, it is important to consider what types of media you will need to access and how to structure access to that media.

LiveCode supports a wide range of media formats. Media can be accessed using the built-in media support, through QuickTime on older Mac OS X systems or via an external library. The advantage of using the built-in support is that you can consistently display or play back the media on all platforms without having to check that any 3rd party component has been installed. Other 3rd party libraries may allow a greater range of media access.

Since each loaded stack file takes up as much memory as the size of all its stacks, it is often advisable to place large, seldom-used objects (such as color pictures, sound, and video) in external files, which can be bundled with your application and loaded into memory only when you need them.

The built-in media support allows you to embed media directly within your stack file, or to reference it externally, storing it in a data folder, online or on a CD.

Embedding Media within your project

Allows distribution of a single-file application for easy, reliable distribution

Requires importing media whenever it is updated

Requires enough memory to load all the media

Allows you to use the built-in editing capabilities directly

Is less practical for creating large themed or localized applications where one set of media is replaced with another set

Referencing Media externally

Requires you to distribute the media files separately

Allows you to edit media files directly, they update automatically

Makes it easy to load and unload media if you need to reduce memory requirements

Requires you to import and export the media to use the built-in editing capabilities

Makes it easy to create themed or localized applications by simply linking to a different directory

Tip: When importing images, use the Image Library, or create a 'library' card that contains all the originals, then reference those objects throughout your project. For more details on referencing images, see the section on the *button* object.

For details of the image formats supported natively, see the section on the *Image* object. For details of the audio formats supported natively, see the section on the *audioClip* object. For details on how to control a movie, see the section on the *Player* object.

How to Use External Files

There are three main ways to use external files:

Keep media such as images, sounds, and video in external files, in the appropriate format, and use referenced controls to display the contents of the files. When you use a referenced control, the image, sound, or video file is loaded into memory only when a card that contains the referenced control is being displayed. The memory needed for the image, sound, or video is therefore only used when actually needed. By using *relative file paths*, you can move the application and its data files to any other system. For more details on using file paths, see the section on *File Name Specifications and File Paths*.

Note: To create a referenced control, use the "New Referenced Control" submenu in the File menu, or create an empty image or player object, then set the object's **fileName** property to a file path for the file you want to reference. For more details, see the section on *Building a User Interface*.

Keep portions of your application in a separate stack file, and refer to the stacks in that stack file as necessary. The **stackFiles** property simplifies referring to external stack files. When you set a stack's **stackFiles** property to include one or more file paths, the stacks at those locations become available to your stack by simply referring to the stacks by name.

Keep portions of your application on a server, and download them using the built in URL commands. For more details on the URL commands, see the section on *Working with Files, URLs & Sockets*.

When to use a Database

You don't need to use an external database to store information for your application. You can store information in stacks, in text files, and in a variety of other files, read it into your application as needed, and modify it. However, as a rule of thumb, we recommend that when you have over two thousand records of information, or you want to allow multiple users to access information at the same time, you consider using a database.

In these circumstances an external database offers many advantages. A database located on your local machine will be fast and efficient at accessing and searching records. A database located on a server can be accessed by more than one user. Depending on the particular implementation, a database may be suitable for constant access by hundreds of users, each receiving and updating data continually. SQL

databases have built-in record-locking capabilities, preventing one user's changes from wiping out another's— a necessity for reliable multi-user databases. Other built-in features work behind the scenes to ensure the data in the database is not corrupted by interference between different users.

SQL databases are also built for speed. When searching tens or hundreds of megabytes or more, the performance of an optimized database will generally be much better than that of a stack doing the same search. Moreover, stacks must be loaded into memory to be searched, and therefore the whole collection of data must fit into local memory.

Finally, if you use an external database, you can put the heavy-duty processing on a server designed for the purpose, while using LiveCode's flexibility to give the user options for selecting data, then presenting it in a usable form.

With LiveCode's built in *Database Library*, your application can communicate with external SQL databases. You can get data from single-user and multi-user databases, update data in them, get information about the database structure, and display data from the database in your stack.

For more details on working with databases, see the *Working with Databases* guide.

LiveCode Memory Limits

The following table details the memory limits for different types of LiveCode components. Please note that these limits refer to the maximums that may be in use at any one time. You can store additional information on disk or in a database and load it when it is required.

A note about entries designated as "Unlimited":

Since each open stack file resides completely in memory, LiveCode stacks (and all structures within a stack) are effectively limited by available memory and by LiveCode's total address space of 4G (4,294,967,296 bytes) on 32-bit systems

Total addressable space	4 GB (on 32-bit systems)
Maximum length of a line in a field	65,536 characters storage No more than 32,786 pixels wide for display
Maximum size of an object	Unlimited
Maximum number of objects in a group	Unlimited
Maximum number of objects on a card	Unlimited
Maximum number of cards in a stack	Unlimited
Maximum number of objects in a stack	Unlimited
Maximum length of object names	65,536 characters
Maximum length of custom property names	255 characters
Maximum length of command or function names	65,536 characters
Maximum size of custom properties	Unlimited
Maximum number of custom properties	Unlimited
Maximum size of a script	Unlimited
Maximum size of other properties	64K
Maximum nesting level in control structures	Unlimited
Maximum level of recursion	Unlimited

Building a User Interface

The user interface for your application is often one of its most important features. Building a clear, logical and aesthetically pleasing user interface will make all the difference to the success of your application.

This chapter discusses how to build a user interface using LiveCode. It tells you how to create and lay out controls, which objects to use and even how to build your own custom objects. We then touch briefly on some tips for good user interface design.

Creating and Organizing Objects

Creating Controls with the Tools Palette

The main tools palette allows you to change between Edit and Run mode, create objects, and edit bitmap images with the paint tools.

Figure 13 - The Main Tools Palette

Run mode	Click to enter run mode. When in run mode, objects receive all the normal messages that drive a LiveCode application. For example, clicking on a button in run mode will cause a mouseUp message to be sent to it and the script will run.
Edit mode	Click to enter edit mode. When in edit mode, objects do not receive messages when you click on them, and you can move, resize or edit the properties for objects. For more details about the run and Edit mode, see the section <i>The 'Edit' and 'Run' Mode</i> .
Widget objects	
Button objects	Drag any object within these areas onto an editable stack to create a new object.
Field objects	Double-click an object to create a object of that type in the center of the front most editable stack. For more details on all of these objects, see the section about each object type later in this chapter.
Scrollbar objects	
Image & Player	
Vector Graphics	Press the grey triangle at the bottom right of the Image & Player area to hide and show this section. Click to choose the type of new graphic you want to create. Use the fill bucket to choose the fill color, the fill pencil to choose the line color, the line thickness menu to choose the line thickness, and the optional shape menu to choose preferences specific to the type of graphic selected. Click and drag in an editable stack to create the new graphic.
Bitmap graphics	The paint tools allow you to edit bitmap graphics that have been imported into LiveCode or create your own. To use them, create an image object and paint within that area, or modify an existing image. You cannot use the paint tools to edit an image that has its filename property set, as that image data is stored outside LiveCode. For more details on working with images, see the section on <i>Images</i> later in this chapter and the chapter on <i>Working with Media</i> .

Tip: You can also create objects using the **New Control** sub-menu in the Object menu, or by typing **create [object type]** in the Message box. For more information on creating objects using the Message Box, see the section on *Building Interfaces Using Scripts*, later in this chapter.

To open a system-standard color chooser, double-click on color chooser popup menus at the bottom of the Vector Graphics or Bitmap Graphics sections of the tools palette.

Alignment & Layering

Figure 14 – Size & Position Inspector

Lock size and position	Locks the object so that its size and position cannot be adjusted interactively with the mouse when in edit mode. This also prevents images, groups and players from automatically resizing to display their entire content whenever the card that they are on is reopened. For more details see the entry for <i>lockLocation</i> in the LiveCode Dictionary.
Width & Height	Allows you to set the width and height of the object(s) currently being operated on by the Property Inspector. Objects are resized from their center. For more details, see the <i>width</i> and <i>height</i> properties in the LiveCode Dictionary.
Fit Content	Automatically sizes the object large enough to display its content. In the case of buttons, the content is the text and any icon. For images, this is the native width and height of the original image before any scaling. For more details, see the entry for <i>formattedWidth</i> and <i>formattedHeight</i> in the LiveCode Dictionary.
Location	Sets the objects position (the center of the object) relative to the top left of the card.
Left, Top, Right & Bottom	Sets the position of one of the object's edges.
Layer	Sets the layer of the object. The buttons with arrows allow you to send an object to the back, move an object back one layer, bring an object forward one layer and bring an object to the front. Layer determines which objects are displayed in front or behind others, as well as the object's number and tabbing order. Note that you cannot relayer objects that are grouped unless you are in edit background mode, or have the <i>relayerGroupedControls</i> set to true. For more details, see the section on <i>Group & Backgrounds</i> . For more details on tabbing order see the section on <i>The Keyboard Focus</i> below.

Use the Align Objects Inspector to resize objects relative to other objects, to reposition objects and/or to relayer objects.

Figure 15 – Align Objects Inspector

To open the Align Objects Inspector, select multiple objects, then open the Inspector and choose Align Objects from the menu at the top of the Inspector. The Align Objects pane will automatically be displayed if you select multiple objects of different types.

Important: The Align Objects Inspector resizes objects relative to each other. Always select the object you want to use as a reference first, then select other objects that you want to make the same as that object next. If you want to distribute objects, select them in the order that you would like them to be distributed.

Equalize	Make objects the same width, height, or have exactly the same rectangle.
Align	Aligns the objects by their left, right, top or bottom edges, or by their horizontal center or vertical center, using the first object selected as a reference.

Distribute	Distributes the objects with an equal difference between them, using the order they were selected. 'First to last' selected will distribute objects evenly between the edges of the first and last objects selected. 'Edge to edge' will distribute the objects so that their edges touch each other exactly. 'Across card' will distribute the objects evenly over the entire card area.
Nudge	Nudge the selected object the number of pixels specified in the center of the arrows. To change the number of pixels, click on the number.
Relayer	'First to last selected' will relayer the objects in the selection in the order they were selected. 'Last to first' will relayer the objects in reverse order. Use these buttons to set the tab order of a set of objects. For more information on the tabbing order, see the section on <i>The Keyboard Focus</i> below.

The Keyboard Focus

The focus is an indication to the user of which control will receive a keystroke. Exactly which objects are capable of receiving the keyboard focus depend on the current operating system, and the properties applied to the control. Edit fields can receive the focus, as can all other objects on Windows and Linux, and many objects on Mac OS.

The order in which the user moves through controls that can receive the keyboard focus is determined by the object's layer. When a card is opened, LiveCode automatically focuses the first object on the card that is capable of receiving the keyboard focus.

You can turn on the ability of an object to get the keyboard focus by checking the *Focus with Keyboard* option in the object's Inspector, or by setting its *traversalOn* property by script.

On some platforms objects give an indication that they have the focus by displaying a border. You can specify whether a border should be displayed by setting an object's Show Focus Border option in the Inspector, or setting its *showFocusBorder* property by script.

Object & Control Types

Stacks - for displaying windows, palettes and dialog boxes

In LiveCode, each window is a stack. This includes editable windows, modeless and modal dialog boxes, and palettes, as well as sub-windows available on some operating systems, such as sheets and drawers.

Tip: If you want to make controls within a stack scale and position automatically when you resize the stack, see the section on the *Geometry Manager*.

This topic discusses the way windows are implemented in LiveCode applications; how to change window appearance; and how to control the behavior of the various window types. This topic does not cover the organization of stacks within a stack file, which is covered in detail in the section *Structuring Your Application*.

Caution: Do not start your stack name with "rev". Stacks with names starting with "rev" are reserved by the LiveCode development environment.

You create a new stack – which can then be displayed in any number of modes (as described below) – by choosing **File -> New Mainstack**. You can edit stack properties by choosing **Object -> Stack Inspector**.

Window Types and the Mode of a Stack

The nature of a stack's window depends on the stack's **style** property and on what command was used to open the stack. You can either specify the window type when opening the stack, or allow the stack's

style property to determine the type of window it is displayed in.

Note: We recommend you specify the mode you want a stack to be opened in as part of the command you use to open the stack, rather than using the style property. This makes it easy to switch a stack between editable mode in the development environment, and for example, a dialog in the runtime environment.

The Standard Window Types

LiveCode windows are usually one of four types: editable or **topLevel** windows, **modal** or **modeless** dialog boxes, or **palette** windows.

Important: You will normally create a new stack and edit it while it is in editable mode. If you want to create a dialog, create the stack like any other stack. Then when opening it, specify in the script that it should be displayed as a dialog, or a palette, etc. The appropriate commands for doing this are detailed with each type of window below.

You can test out these commands as you work on your window layout and scripts, using the *Message Box* (see section of the same name), or using the *window context menu* (see *Modal Dialog Boxes*, below). For more details on writing scripts in general, see the section on *Coding in LiveCode*.

Most windows are editable or **topLevel** windows, and this is the default mode for LiveCode stacks. If you open a stack using the **go** command (without specifying a mode), or using the Open Stack menu item, then the stack is displayed as an editable window unless its **style** property specifies another window type.

Editable Windows - for documents

An editable window has the appearance and behavior of a standard document window. It can be interleaved with other windows, and you can use any of LiveCode's tools to create, select, move, or delete objects in the stack.



Figure 16 - Editable Document Windows on Multiple Platforms

To display a stack in an editable window, you use the **topLevel** or **go** commands:

```
topLevel stack "My Stack"
go stack "My Stack" - "topLevel" is the default mode
go stack "My Stack" as topLevel
```

Stacks whose **style** property is set to "topLevel" always open as editable windows, regardless of what command you use to open them.

Note: If the stack's **cantModify** property is set to true, the window retains its standard appearance, but tools other than the Browse tool can no longer be used in it. In other words, every tool behaves like the Browse tool when clicking in an unmodifiable stack's window.

Modeless Dialog Boxes - for alerts and settings

Modeless dialog boxes are similar to editable windows. Like editable windows, they can be interleaved with other windows in the application. Their appearance may differ slightly from the appearance of

editable windows, depending on the platform.

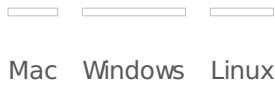


Figure 17 – Modeless Dialog Boxes on Multiple Platforms

Like unmodifiable stacks, modeless dialog boxes do not allow use of tools other than the Browse tool. Use modeless dialog boxes in your application for windows such as a Preferences or Find dialog box, that do not require the user to dismiss them before doing another task.

To display a stack in a modeless dialog box, you use the **modeless** or **go** commands:

```
modeless stack "My Stack"
go stack "My Stack" as modeless
```

Stacks whose **style** property is set to "modeless" always open as modeless dialog boxes, regardless of what command you use to open them.

Modal Dialog Boxes - for alerts and settings

A modal dialog box is a window that blocks other actions while the window is displayed. You cannot bring another window in the application to the front until the dialog box is closed, nor can you edit the stack using the tools in the Tools palette. While a modal dialog box is being displayed, the handler that displayed it pauses until the dialog box is closed.



Figure 18 – Modal Dialog Boxes on Multiple Platforms

Modal dialog boxes do not have a close box. Usually, they contain one or more buttons that close the window when clicked.

Important: If you mistakenly open a modal dialog box without having included a button to close the window, use the contextual-menu shortcut (Control-Shift-Right-click for Unix or Windows, Command-Control-Shift-click for Mac OS) to display a context menu. Choose *toplevel* to make the stack editable.

To display a stack in a modal dialog box, you use the **modal** command or **go** commands:

```
modal stack "My Stack"
go stack "My Stack" as modal
```

Stacks whose **style** property is set to "modal" always open as modal dialog boxes, regardless of what command you use to open them.

Palettes - for accessory and tool windows

A palette has a slightly different appearance, with a narrower title bar than an editable window. Like dialog box windows, a palette does not allow use of tools other than the Browse tool.

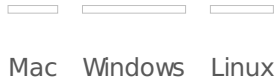


Figure 19 – Palette Windows on Multiple Platforms

A palette floats in front of any editable windows or modeless dialog boxes that are in the same application. Even when you bring another window to the front, it does not cover any palettes.

Note: On some Linux systems, palettes have the same appearance and behavior as ordinary windows and do not float. On Mac OS X systems, palette windows disappear when their application is in the background and another application is in front.

To display a stack in a palette, you use the **palette** command or **go** commands:

```
palette stack "My Stack"
go stack "My Stack" as palette
```

Stacks whose **style** property is set to "palette" always open as palettes, regardless of what command you use to open them.

Ask Question Dialog - for asking a question

The ask question dialog is a special type of window that is designed to make it easy to ask the user a question. It includes special syntax for opening the dialog with question text and returning any answer to the script that called it. You can also specify the window title, as well as an icon to be displayed in the window. The font, object positions, button order and icon will *automatically change* to reflect the operating system. However, if you do require more flexibility than is provided in this dialog, you should create your own modal dialog box instead (see above).



Figure 20 – Ask Question Dialog Boxes on Multiple Platforms

To display the ask dialog, use the following commands:

```
ask "What is your name?"
ask question "What is the answer?" titled "Question"
```

Tip: You can try this example out right now by coping and pasting it into **Tools -> Message Box**, then pressing return.

The word question specifies the icon, and may be any of question | information | error warning.

To change the miniature application icon shown in the Mac OS X platform picture above, see the entry for gREVAplcon in the *LiveCode Dictionary*.

The result is returned in the *special variable* it.

```
if it is "Joe" then doSomething
```

For complete details on the syntax, see the ask command in the *LiveCode Dictionary*.

Figure 21 – Icons for Ask and Answer Dialogs on Multiple Platforms

Answer Alert Dialog - for displaying a dialog

Like the ask dialog box above, the answer dialog is a special dialog that has been designed to make it easy to display information in a dialog on screen and optionally allow the user to make a choice from a list of up to seven choices. The answer command opens the dialog, lets you specify the text and the button choices, and returns the button click to the script that called it. You can also specify the window title, as well as an icon to be displayed in the window. As with the ask dialog, the font, object positions, button order and icon will *automatically change* to reflect the operating system. However, if you do require more flexibility than is provided in this dialog, you should create your own modal dialog box instead (see above).



Figure 22 – Answer Dialogs on Multiple Platforms

```
answer "Hello World!"

answer question "What city is the capital of Italy?" with \
  "Paris" or "London" or "Rome" titled "Multiple Choice"
```

The result is returned in the *special variable* it.

```
if it is "Rome" then
  answer information "That was the correct answer."
end if
```

For complete details on the syntax, see the answer command in the *LiveCode Dictionary*.

The answer dialog is implemented internally as a stack attached to the LiveCode IDE. For details on customizing the IDE, see the section on *Editing the LiveCode User Interface*.

Tip: If you're not sure what a stack's name is, you can use the **mouseStack** function to find out. Enter the following in the Message Box (Tools -> Message Box), then move the mouse pointer over the stack window and press Return: `put the mouseStack`

File Selector Dialogs

The file selector dialogs allow you to display the system standard dialogs. These dialogs allow the user to select a file or a set of files, select a directory, or specify a name and location to save a file.

The syntax for bringing up the file selector dialogs mirrors the syntax for the alert and question dialogs detailed above. However, unlike these dialogs, the file selector dialogs are displayed using the system standard dialogs where available. The notable exception at the time of this writing is Linux platform, where a built-in dialog is used instead due to more limited OS support. (You can force the other platforms to use this built-in stack based file selector by setting the *systemFileSelector* global property.)

Figure 23 – Answer File dialog for selecting a file

```
answer file "Select a file:"

answer file "Select an image file:" \
  with type "Movies|avi" or type "All Files|"
```

The file path to the file selected by the user is returned in the *special variable* *it*. If the user cancelled the dialog, the special variable *it* will be empty and "cancel" will be returned by *the result* function.

For complete details on the syntax, see the **answer file with type** command in the *LiveCode Dictionary*.

Figure 24 – Ask File dialog for saving a file

```
ask file "Save this document as:" with "Untitled.txt"
answer file "Select an image file:" \
  with type "Text Files|txt" or type "All Files|"
```

The file path to the file to be saved is returned in the special variable *it*. If the user cancelled the dialog, the *it* variable will be empty and cancel will be returned by *the result* function.

For complete details on the syntax, see the **ask file with type** in the *LiveCode Dictionary*.

Figure 25 – Answer folder dialog for choosing a directory

```
answer folder "Please choose a folder:"

answer folder "Please choose a folder:" with "/root/default folder"
```

The file path to the folder selected by the user is returned in the special variable *it*. If the user cancelled the dialog, the *it* variable will be empty and "cancel" information will be returned by *the result* function.

For complete details on the syntax, see **answer folder** in the *LiveCode Dictionary*.

Color Chooser Dialog

The **answer color** dialog allows you to display the operating system's standard color picker dialog.



Figure 26 – Answer color dialog for choosing a color

answer color

The color chosen is returned in the special variable **it**. If the user cancelled the dialog, it will be empty and "cancel" will be returned by *the result* function.

For complete details on the syntax, see **answer color** in the *LiveCode Dictionary*.

Printer Dialogs

The printer dialogs allow you to display the standard printer and page setup dialogs.



Figure 27 – Answer printer for standard printer dialog

Use the **answer printer** command to display a standard printer dialog prior to printing. If the user cancels the dialog, "cancel" will be returned by *the result* function.

Alpha Blend Windows – for Enhanced Tooltips and Multimedia

Use the *Shape* option in the *Stack Inspector* to set a stack's **windowShape** property to the transparent, or *alpha channel* of an image that has been imported together with its alpha channel (i.e. in either PNG or GIF format). This allows you to create a window with "holes" or a window with variable translucency. You can apply a shape to any type of stack, regardless of the mode it is opened, allowing such a window to block as a dialog, float as a palette, etc.

Important: The border and title bar of a stack are not shown if the stack's **windowShape** is set. This means you will need to provide methods of dragging and closing the window if you want the user to be able to do these tasks.



Figure 29 – Window with alpha mask applied

You can change the **windowShape** property dynamically by script to a series of images to create an animated translucent window.

System Palettes – for utilities floating above applications

A system palette is like a palette, except that it floats in front of all windows on the screen, not just the windows in its application. Use system palettes for utilities that you want the user to be able to see and access in every application.



Figure 30 – System Window floating above other applications

To display a stack in a system palette, you turn on the check box in the Stack Inspector "Float Above Everything". For more details on this feature, see the entry for `systemWindow` in the *LiveCode Dictionary*.

Using this feature overrides the stack's **style** or **mode**.

The system palette style is currently not supported on Linux & Unix.

Stack menus - for displaying non-standard menus

Note: Usually a menu in a LiveCode application is implemented as a button. We recommend that menus are implemented using buttons, as these will automatically be drawn with the native theme on each platform. For more details, see the section on Menus below

It is also possible to display a stack as a *pulldown*, *popup*, or *option* menu. Stack menus are used when a menu needs to contain something other than just text. For example, a popup menu containing a slider, or an option menu consisting of icons instead of text, can be implemented as a stack menu.

To display the stack menu, you create a button and set its **menuName** property to the stack's name. When the user clicks the button, the stack is displayed with the behavior of a menu. Internally, the menu is implemented as a window, and you can use the **popup**, **pulldown**, or **option** command to display any stack as one of these menu types.

Stack Decorations - for Window Appearance

Stack decorations allow you to specify how the title bar and border of a window will be drawn. You can access the stack decorations options in **Object -> Stack Inspector**.

Apart from the differences in appearance between different window modes, the appearance and functionality of a window can vary depending on the stack's properties and the platform it is being displayed on. A window's title bar displays the window's title, as well as a close box, minimize box or collapse box, and maximize box or zoom box.



Figure 31 – Window Decorations on Windows and Mac OS X

Note: On Mac OS X, a stack's **shadow** property controls whether the stack window has a drop shadow. On OS X systems, you can set this property to false to create a window with no shadow.

The properties above can also be set by script, for more details see the **decorations** property in the *LiveCode Dictionary*.

While the stack's mode is separate from its **decorations**, the mode may affect whether these properties have an effect. If the **decorations** property is set to "default", it displays the appropriate decorations *for the current window type* on the current platform.

Button Controls - for performing actions

A button is a clickable object that is typically for allowing a user to perform an action by clicking.





Figure 32 – Button Objects on Multiple Platforms

Check boxes and radio buttons are used to allow the user to make choices. Radio buttons are used when only one option for a set of options may be selected at any time. Check boxes are used where some options may be turned on and others may be off.

Note: LiveCode will automatically enforce the rule of highlighting one radio button at a time if you place the radio buttons together in a group. For more details on groups, see the section on *Groups and Backgrounds*.



Figure 33 – Check Boxes and Radio Buttons on Multiple Platforms

All button objects in LiveCode are highly flexible and customizable. Common settings include the ability to show and hide the border or fill, and to display an icon.

Icons allow you to provide a wide range of functionality. For example you can create a roll over effect by setting a hover icon. Or you can create a custom check box by setting an icon and a highlight icon – doing so will replace the system standard check box and display your icon in each state depending on whether the button has been pressed or depressed.



Figure 34 – Custom Buttons for Multimedia

Important: Button icons are not limited in width or height. They can be animated by using an animated GIF. In fact, an icon can reference **any** image contained within your LiveCode stack file. Referencing an image in this way saves disk space and allows you to update all icons in your stack by updating a single image. See the chapter *Working with Media* for more information.

Text Field Controls - for displaying or entering text

Fields allow you to display text. Fields can optionally allow the user to edit the text. Fields support multiple fonts, styles and colors, images and a subset of basic HTML tags. Fields can be linked to a database using the Database Query Builder or by directly accessing a database with the database library. They can display and render XML using the XML library. List fields allow the user to select one or a set of choices. Table fields allow display of data similar to a spreadsheet. Other types of field can easily be created, including tree views, or any hybrid between these types, with a little scripting. At the time of this writing there is also a 3rd party library available that allows you to host a web browser within an object and control it by script.

Figure 35 – Field Controls

List and Table Field Controls

List fields allow you to display a set of choices. Users cannot edit list fields. You can specify whether the user is allowed to make a single selection or multiple selections.

Figure 36 – List Fields

Table Field Control - for displaying a table

Table fields allow you to display data in cells, and optionally allow the user to edit the cells. Table fields are ideal for displaying basic tabular data. For more complex data display requirements we recommend you use the Data Grid control.

Figure 37 – Table Field

Data Grid Control - for presenting data

Data Grids allow you to display data in both grid and form modes. You can customize a data grid to include custom layouts that include any other LiveCode object. Data grids can provide a view into a data source, allowing you to display large data sets. For full documentation on using data grids, see the [online documentation](#)

Figure 38 – Data Grid

Cards

Each stack contains one or more separate screens of controls. Each screen is known as a card. Each card can have an entirely different appearance, or all the cards in a stack can share some or all elements, by using *shared groups*, known as *backgrounds* (see below).

Choosing **Object -> New Card** will create a new card within the current stack. The new card will either be completely blank, or will contain any shared groups from the previous card.

Groups & Backgrounds - for organizing, and sharing controls

Groups, LiveCode's most versatile object type, are used for several purposes: radio button clusters; menu bars; for creating scrollable object areas within cards; and as backgrounds for displaying sets of objects that are shared between cards. Groups can also be used for creating a simple card and stack database, by holding fields that contain a different record on each card.

What Is a Group?

A group is a single object that holds a set of objects. Objects are grouped by selecting the controls you

want to include in the group, then using the **group** command or choosing **Object -> Group Selected**.

Once you've created the group, it becomes an object in its own right. You can select, copy, move, and resize the group, and all the objects in the group come with it. The objects in the group maintain their own identities, and you can add objects to the group or delete them, but the objects are owned by the group instead of the card.

A group has its own properties and its own script. Groups can be any size, can be shown or hidden, and can be moved to any location in the stack window, just like any other control. Like other controls, groups can be layered in any order with the other controls on the card. Groups can also display a border around a set of objects.

Figure 39 – Group of radio buttons with group title and border

Unlike other controls, however, groups can appear on more than one card. You place a group on a card using the **place** command or the **Place Group** submenu in the **Object menu**.

Important: A group that is shared between cards appears at the same location on each card. A change made to the position of a shared group on one card is reflected on all the other cards that share the group.

Groups and Backgrounds

Both the term **group** and the term **background** can be used to refer to groups. The terms are interchangeable in some circumstances and mean different things in others. The differences are explained in more detail below.

In general, the term **group** refers to groups that are placed on a card, while the term **background** refers to all the groups in a stack that are available for use as *backgrounds* (see below). The expression

`the number of groups`

evaluates to the number of groups on the current card. The expression `the number of backgrounds` evaluates to the number of background groups in the current stack, including groups that are not placed on the current card.

Tip: When you refer to a group by number, if you use the word **group**, the number is interpreted as referring to the groups on the referenced card, in order by layer. If you use the word **background**, the number is interpreted as referring to the groups in the stack, in the order of their creation.

For example, the expression `the name of group 1` evaluates to the **name** of the lowest-layered group on the current card, while the expression `the name of background 1` evaluates to the **name** of the first group that was created in the stack--whether or not that particular group is placed on the current card, or appears on any card at all.

The term **background** can be also used to refer to the set of cards that share a particular group. The following statement goes to the third card on which the group named "Navigation" is placed:

`go card 3 of background "Navigation"`

Nested Groups

LiveCode supports nested groups (one group containing another). Since a group is itself a control, it can be contained in another group.

Creating a nested group is just like creating a group: select the controls you want to group (including the existing group), then choose **Object -> Group Selected**. The existing group is now a member of the new group.

Selecting and Editing Groups

To select a group, simply click on one of the objects that is contained within it. This will select the group.

If you want to select an object within the group, instead of the group itself, there are two ways to do so. You can turn on the **Select Grouped** option on the toolbar or in **Edit -> Select Grouped Controls**. This causes groups to be ignored when selecting objects, allowing you to select objects inside a group as if the group didn't exist. You can enter this mode by script by toggling the **selectGroupedControls** global property.

Alternatively you can go into edit group mode, a special mode that only displays the objects within that group. Select the group, then press **Edit Group** on the toolbar or choose **Object -> Edit Group**. When you have finished, choose **Object -> Stop Editing Group**. You can toggle this mode programmatically by using the commands **start editing** and **stop editing**.

Tip: If a group's border has been set, an outline appears at the group's edges. However, clicking within or on the border does not select the group. To select the group, you must click one of its controls.

Placing and Removing Backgrounds

Once you create a group, you can display it on any or all cards in the stack. First, ensure that the group's Behave as Background option has been set in the Inspector. Then navigate to the card you want to place the group on and choose **Object -> Place Group** to place an instance of a particular group on the current card. You can control these features from script by using the **backgroundBehavior** property and **place** command.

Note: When you create a new card, if there are any groups on the current card whose Behave as Background has been set, they are automatically placed on the new card. To make it easy for all the cards in a stack to share a single group, create the group on the first card and set this property to true, before you create any other cards.

To remove a group from the current card without deleting it from the stack select the group and choose **Object -> Remove Group**. The group disappears from the current card, but it's still placed on any other cards that share the group. You can remove a group by script using the **remove** command.

Tip: You can use the **start editing** command from the Message Box to edit a group that has not been placed on any card. Since the group is not on any card, you must refer to it using the term "background" instead of the term "group".

You can completely delete a group in the same way as you delete any other object, by selecting the group and choose **Edit -> Clear** or pressing backspace.

Important: Deleting a background group removes it from all the cards it appears on, and from the stack itself.

To dissolve a group back into its component controls, select the group and choose **Object -> Ungroup**. You can ungroup a group by script using the **ungroup** command. Ungrouping deletes the group object and its properties (including its script) from the stack, but does not delete the controls in it. Instead, they become card controls of the current card. The controls disappear from all other cards the group is on.

Note: If you ungroup a group, then select the controls and regroup them before leaving the current card, the group is restored as it was. However, leaving the current card makes the ungrouping permanent and deletes the group from all other cards it was on.

Groups and the Message Path

For details on how groups and backgrounds fit into the Message Path, see the section on *Groups, Backgrounds & The Message Path*, below.

Graphics, Images, Players, Audio & Video Clip Objects - for multimedia

LiveCode supports a wide range of media formats, allowing you to produce rich media applications. The image object allows you to import or reference images, manipulate images by script or interactively with the paint tools, and save them out in different formats with variable compression options. Support extends to alpha channelled PNG images and animated GIF images. Images can be imported and reused within a stack to create custom or interactive interface elements. To learn how to work with these objects in a script, see the section on *Working with Media*.

Image formats supported include GIF, JPEG, PNG, BMP, XWD, XBM, XPM, or PBM, PGM, or PPM files. On Mac OS systems, PICT files can also be imported (but they cannot be displayed on Unix or Windows systems). For full details on each of these formats, see the section on *Working with Media*.

You can import images using **File -> Import as Control -> Image File**. You can reference an image using **File -> New Referenced Control -> Image file**.

Paint tools can only be used on images that have been imported as a control. See the section on *Using the Paint Tools* for details on how to use the paint tools.

Figure 40 – Image, Graphic and Player objects

Vector graphics can also be created and manipulated with the graphic tools and by script. LiveCode supports paths with variable fills, gradients, blended and antialiased graphics. At time of this writing there is also a 3rd party library that allows the import and export of SVG format graphics. Use graphic objects to create interactive interfaces, graphs, charts or games.

Use the player object to display and interact with any media formats supported by AVFoundation on Mac OS X and DirectShow on Windows. LiveCode allows you to turn on and off tracks within a movie and set callback messages that trigger scripts at specific points in the movie, and stream movies from a server. At time of this writing there is also a 3rd party library that allows you to edit and save movies by script.

The Audio Clip & Video Clip objects allow you to embed audio or video clip data within a stack. They do not have any visual representation and can be accessed by script or in the Application Browser. Players, Audio & Video clips are covered in their respective sections within the chapter on *Working with Media*.

Menu Controls - for displaying choices

Menus are used to display a list of choices. The *pulldown* menu displays a standard *pulldown* menu, and can be automatically inserted into the main menu bar on Mac OS systems. The *option* menu allows a choice from a list. The combobox allows the user to type an *option* or choose from a list. Popup menus can be displayed underneath the cursor and used to provide context sensitive options anywhere in your application. For more information on working with *pulldown* menus within the main menu bar, see the section on the *Menu Builder* below.

Menu contents can be defined using a list of item names and special characters to indicate where shortcuts and checkmarks should be placed. This is the most common type of menu and is known as a *contents menu*. LiveCode will automatically draw menus defined as *contents* menus using the native

system look on each platform. When you choose an item from a contents menu, LiveCode will send a **menuPick** message along with the name of the item chosen.

Alternatively, menus may be constructed from a *stack panel*, giving you complete control over the menu contents and allowing the display of any object type of functionality. When you choose an item from a *stack panel* menu, the individual object within the menu will receive a **mouseUp** message. Note that panel menus cannot be displayed within the main menu bar on Mac OS systems.

For more details on working with and scripting menus in general, see the section *Working with Menus* in the *Programming a User Interface* guide.

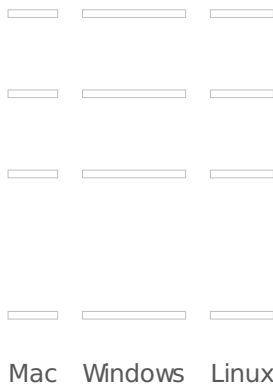


Figure 41 – Menu Controls

The cascade menu is a special type of control that is only used when building a stack panel menu. Sub-menu items can be created in list-based content menus without using this object.



Figure 42 – Tab Menu on Windows

The tabbed panel is a type of menu in LiveCode. You can specify a list of tabs to be displayed and receive a **menuPick** message when the user clicks on a tab in the same way as other menus. There are two common techniques for implementing a tabbed interface: group the objects for each tab together and show or hide the appropriate group when you change tab; or place the tab object into a group which is then placed as a background on multiple cards.

Other Controls

Scroll bars can be used as a *progress* bar to display a value, a *slider* to allow the user to choose a value, or to scroll objects. (Note that you don't need to use a scrollbar object with fields or groups as these can display a built-in scroll bar.) Sliders and scrollbars can be displayed both horizontally and vertically – to display vertically, resize so that the height is greater than the width.



Figure 43 – Scrollbars

Using the Menu Builder

The menu builder allows you to create and edit a standard menu bar that will work correctly regardless of the platform you intend to deploy on. On Windows and Unix, menus built with the Menu Builder will appear in the top of the window. On Mac OS they will be displayed in the main menu bar. It is also possible to generate a menu bar by script. For more details, see the section on *Programming a User Interface*.

Choose **Tools -> Menu Builder** to open the Menu Builder.

Figure 44 – Menu Builder

Preview	This option allows you to preview your menu bar in the main menu bar. It only applies on Mac OS systems where the option Set as Menu Bar on Mac OS has been turned on.
Menu bar settings	<p>> Important: To bring back the LiveCode development menu bar when working with a stack that has this option turned on, click on a LiveCode IDE window such as the Tool bar.</p> <p>This area specifies the main settings for your menu bar. Use the <i>New</i> button to create a new menu bar in the current top most editable stack. Enter the name for your menu bar in the text area. <i>Delete</i> will permanently delete your menu bar. Use the <i>Edit</i> button to load an existing menu bar from the top most editable stack to edit in the area below.</p>
Menu edit area	<p>Select a menu to work on from the scrolling list. At a minimum your application should have a File, Edit and Help menu. These menus are created for you automatically when you create a new menu bar. To create a new menu, move the orange <i>divider bar</i> to the position in the menu bar you want to create the new menu, and press <i>New Menu</i>. Disable the currently selected menu by checking <i>Disabled</i>. Choose the keyboard shortcut (the portion of the name that is underlined), using the <i>Mnemonic</i> popup menu (Windows, Linux & Unix only). To move a menu in the list, select it then press the <i>up or down arrows</i> (to the right of the name area).</p>
Menu content area	<p>Select a menu item to work on from the scrolling list. To create a new menu item, move the orange <i>divider bar</i> to the position in the menu bar you want to create the new menu, and press <i>New Item</i>. Disable the currently selected item checking <i>Disabled</i>. Choose the keyboard shortcut (the portion of the name that is underlined, for use when the menu is open), using the <i>Mnemonic</i> popup menu (Windows, Linux & Unix only). To move a menu item up or down the list, select it then click the <i>up or down arrows</i> (to the right of the name area). To move items into a <i>submenu</i>, click the <i>Right</i> arrow, or click the <i>Left</i> arrow to move a submenu item back into the main menu bar. To insert a <i>divider</i>, position the orange divider bar where you want the divider, then click the blue divider button (top right). To make the menu item a <i>Checkbox</i> or <i>Diamond</i> option, choose the appropriate option from the <i>Mark</i> popup menu. To create a control key <i>shortcut</i> for the item, click the Shortcut check box and enter the letter you want to use for the shortcut. To understand the <i>symbols</i> that are created next to the menu items, see the section on <i>Menu Bars</i> within the chapter on <i>Programming a User Interface</i>.</p>
Scripting	<p><i>Edit Script</i> opens the Code Editor for the currently selected menu. <i>Auto Script</i> places a blank script within the currently selected button with spaces to insert actions for each of the menu items within that item. We recommend you press Auto Script before pressing Edit Script when you create a menu.</p>

Using the Geometry Manager

Use the Geometry Manager to specify how objects should be scaled and positioned when the user resizes the window.

Figure 45 – The Geometry Manager

Scale or Position Selector

Choose whether you want the control to be *scaled* or *positioned* when the stack is resized. Scaling will change the dimensions of the control as the stack resizes. Positioning will move the control, it does not change its dimensions. Note that it is possible to scale an object in the horizontal plane and have it position in the vertical. Select Scale then set the options for one axis in the Linking area. Then select Position and set the options for the other axis. If you set options in both the Scale and Position modes for both axis, the Scaling options will be ignored.

Use the linking area to specify the *relationship* between the control and the window, or other controls as the stack is resized. In Scaling mode you can link each edge of the object to the window or another object. In positioning mode, you can only link the X and Y axis of an object. Click the gray bars to create a link. A single click results in an absolute link, a second click will create a relative link. An absolute link will keep the object the current number of pixels away from what it is being linked to. For example, if you link the right edge to the edge of the window and the edge is currently 10 pixels away from the edge of the window, whenever the stack is resized the edge will remain exactly 10 pixels away. However if you use a relative link the distance will be calculated as a percentage of the total width of the card. The object will remain the same percentage away from the edge of the card, the exact number of pixels will vary.

Linking area

When linking to another control, be sure to link to a control that is moved relative to the window, or by a script (e.g. in a `resizeStack` handler).

When using the Geometry Manager with an existing `resizeStack` handler, be sure to pass the `resizeStack` message, otherwise the Geometry Manager will not be able to take effect.

To force the Geometry to manually update, call **revUpdateGeometry**.

> **Tip:** You can use the Geometry Manager to scale objects with a "divider" bar. Create and script the bar to move, then link the edges of the controls to it, then call `revUpdateGeometry` each time it moves to have the objects scale automatically.

Clipping settings

Turn on Prevent Object Clipping Text to prevent the control getting too small to display its label when the window is resized. If the control is a field, you can also turn on the option to display scroll bars if the text within the field does not fit.

Limit settings

Allows you to set the minimum and maximum possible widths and heights for the object.

Remove All

Removes all Geometry settings from the control. Use this option if the settings you have applied do not give the desired effect and you want to start over.

10 Tips for Good User Interface Design

If you are creating a simple utility for yourself, a handful of other people, or as a research project, the design of the interface is less important. However, if you are creating software for a wide group of end users, you should take time to carefully design the user interface. As computer software has become

more mature over the past couple of decades, user expectations of what their software experience should be have increased. Increasingly, users expect clear, uncluttered interfaces which are visually appealing. Getting this right is part art and part science. It is beyond the scope of this manual to give detailed instruction on this topic. There are many good books and resources devoted to this evolving area. However we thought it would be useful to give you our top 10 tips for good user interface design.

- Less is more

Don't use three buttons where one would do. The fewer choices a user has to make the easier your software will be to learn.

- Design to communicate

Good design supports whatever you are trying to communicate.

- Get the defaults right

Providing preferences is great for power users. Just remember that the majority of your users will never adjust the defaults. So focus on getting the default behaviors right before you start to add preferences.

- Layout

A clean consistent layout helps to convey a feeling of professionalism and make your software useable. If you have used a button that is 20 pixels wide in one part of your interface then use the same size of button elsewhere. Line your objects up carefully.

- Screen resolution

Consider what screen resolution you are designing for and whether or not you want your interface to be resizable early on. This decision will have a big impact on your design.

- Consider the program flow

If your program performs a complex task with many options, consider creating a wizard style interface which guides the user step by step through the task. That way they are never faced with a screen covered with dozens of options and no idea what to do next.

- Test it on real users

Bring in a group of people and watch them use the program. Don't interfere or help them out, just get them going and take notes. If you don't have a budget for expensive user testing labs, this process can be as simple as gathering a group of students and offering them free pizza. You'll get a ton of useful feedback that will really help make your program easy to use.

- Don't use square wheels

The various types of widget and control type available have built up a meaning for users over years of use. Don't use something that has a well understood function to perform a different sort of task. If you need a widget that does something new, build something new.

- Skinning

If you're producing a custom skin for your application, a good rule of thumb is either make it entirely custom or use standard OS widgets. An OS native button can look very out of place in the middle of a carefully designed custom skin.

- HCI guidelines for the three main platforms

Each of the platforms that LiveCode supports has its own set of User Interface Guidelines. We recommend you take time to familiarize yourself with them.

Apple Human Interface Guidelines:

http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIIGuidelines/XHIIGIntro/chapter_1_section_1.html

Windows Vista User Experience Guidelines:

<http://msdn2.microsoft.com/en-us/library/aa511258.aspx>

GNOME Human Interface Guidelines 2.0:

<http://library.gnome.org/devel/hig-book/stable/>

KDE Human Interface Guidelines:

http://wiki.openusability.org/guidelines/index.php/Main_Page

Coding in LiveCode

Writing code is how you give your application functionality. Writing the right code means your application will do what you want it to do. Fortunately, LiveCode's built in high level language, makes this task easy. The English-like syntax is easy to read and write. This chapter walks you through writing LiveCode.

The Structure of a Script

What is a Script

Every object in LiveCode can contain a script, which tells it what to do. You edit the script of an object using the Code Editor (see *The Code Editor*). A script is organized into a set of individual *message handlers*, each one of which can respond to a different event (see *Messages*) or contain a specific piece of functionality. Scripts can send messages to each other. In a well organized application, a script will regularly pass data between different types of message handlers which have been grouped and organized to deliver functionality, with a minimum amount of duplication. A script can contain sections of code that is commented – remarks that are intended for a human to read and are not executed. Technically a script is simply another object property, so one script can set another – within certain limits.

The Types of Handler

A *handler* is a complete section of code. Each handler can be executed independently. There are four types of handler: commands (message handlers), functions, getProp handlers and setProp handlers.

Message Handlers

Each *message* handler begins with the **on** control structure followed by the name of the message that this handler responds to. The handler ends with the **end** control structure and the name of the message. Message handlers look like this:

```
on mouseUp
  beep
end mouseUp
```

A message handler is executed when the object whose script contains the handler receives the message. This example handler responds to the **mouseUp** message (see *Messages*).

Function Handlers

Each function handler begins with the **function** control structure followed by the name of the function that this handler computes. The handler ends with the **end** control structure and the name of the function. Function handlers are typically called by another handler and return a value using the **return** control structure. Function handlers look like this:

```
function currentDay
    return item 1 of the long date
end currentDay
```

A function handler is executed when a handler in the same script (or one in an object lower in the *message hierarchy*) calls the function. This example handler returns today's name.

GetProp Handlers

Each **getProp** handler begins with the **getProp** control structure followed by the name of the custom property that this handler corresponds to. The handler ends with the **end** control structure and the name of the property. **getProp** handlers look like this:

```
getProp myCustomProperty
    return the scroll of me + 20
end myCustomProperty
```

A **getProp** handler is executed whenever the value of the corresponding custom property is requested by a LiveCode statement. You can write a **getProp** handler for any custom property of the object or another object lower in the message hierarchy. For more information, see the section on *Custom Properties*.

SetProp Handlers

Each **setProp** handler begins with the **setProp** control structure followed by the name of the custom property that this handler corresponds to. The handler ends with the **end** control structure and the name of the property. **setProp** handlers look like this:

```
setProp myCustomProperty newSetting
    set the hilite of me to true
    pass myCustomProperty
end myCustomProperty
```

A **setProp** handler is executed whenever the value of the corresponding custom property is changed by the **set** command. You can write a **setProp** handler for any custom property of the object or another object lower in the object hierarchy. For more information, see the section on *Custom Properties*.

Comments

Comments are remarks that are intended for a human to read and are not executed. For some recommendations on what sort of comments to include and when, see the section on *Good Design Recommendations*. Comments can be placed either in a handler, or outside any handler.

Any line (or portion of a line) that starts with two dashes (--) or a hash mark (#) is a comment. Placing these characters at the beginning of a line is called "commenting out" the line.

```
on mouseUp -- this part is a comment
    beep
    -- and this is a comment too
end mouseUp
```

You can temporarily remove a statement, or even an entire handler, by commenting it out. To comment out several lines at once, select them and choose **Script -> Comment**.

Since comments are not executed, you can place them anywhere in a script--inside a handler or outside all handlers.

Comments that start with -- or ## only extend to the end of the line. To create a multiple-line comment, or *block comment*, surround it with /* and */ instead:

```
on openCard
    /* This is a multiple-line comment that might
    contain extensive information about this handler,
    such as the author's name, a description, or the
    date the handler was last changed. */
    show image "My Image"
    pass openCard /* You can also make one-line comments */
end openCard
```

Block comments are handy when you want to temporarily remove a section of code while debugging a handler. You can place the characters "/*" at the start of the section, and "*/" at the end, to prevent the section from being executed.

Compiling a Script

A script is compiled when you change the script either by clicking Apply in the Code Editor (or, if altering a script from another script, using the **set** command). During compilation, the entire script is analyzed.

If a compile error is found when a script is being compiled, the entire script is unavailable for execution until the error is corrected and the script is re-compiled. This applies only to compile errors. If an execution error occurs during the execution of a handler, it does not affect the ability to use other handlers in the same script. For more information on dealing with errors, see the section on *Debugging*.

You cannot change a script while a handler in it is executing, because what is executed is the compiled version, not the text in the **script** property.

Summary

Each object has a script, which can be empty or can contain one or more LiveCode handlers. You change a script using the Code Editor, or by setting the object's **script** property.

A script can contain four kinds of handlers: commands, function handlers, setProp handlers, and getProp handlers.

A comment is a part of the script that is not executed. Comments start with -- or #.

If a script contains a compile error, none of its handlers can be used until the error is fixed.

Events

LiveCode is based upon events. Every action a script takes is triggered by an event, which is sent in the form of a message.

What Causes Messages to be Sent

Messages are sent by *events*. Events include user actions (such as typing a key or clicking the mouse button) and program actions (such as completing a file download or quitting the application). LiveCode watches for events and sends a message to the appropriate object when an event occurs.

Note: When a tool other than the Browse tool is active, the development environment traps the built-in messages that are normally sent when clicking (such as **mouseDown** and **mouseUp**). This is so that, for example, you can use the Pointer tool to select and move a button without triggering its **mouseUp** handler.

These messages are referred to as built-in messages, and include **mouseDown**, **mouseUp**, **keyDown**, **openCard**, and all the other messages described in the LiveCode Dictionary.

LiveCode also sends a message whenever a handler executes a *custom* command (see *Sending Messages*). However, *built-in* commands are executed directly by the engine and don't result in sending a message. Similarly, LiveCode sends a function call whenever a handler calls a *custom* function, a **setProp** trigger whenever a handler sets a *custom* property, and a **getProp** call whenever a handler gets the value of a custom property.

Responding to Events

To respond to a message, you write a message handler with the same name as the message. For example, to respond to a **keyDown** message sent to a field (which is sent when the user presses a key while the insertion point is in the field), place a **keyDown** handler in the field's script:

```
on keyDown theKey -- responds to keyDown message
    if theKey is a number then beep
end keyDown
```

The Message Path

The *message path* is the set of rules that determine which objects, in which order, have the opportunity to respond to a message. The message path is based on the *object hierarchy*.

The Object Hierarchy

Each LiveCode object is part of another object, of a different object type. For example, each card is part of a stack, each grouped control is part of a group, and so on. This *object hierarchy* defines the ownership and inheritance relationship between objects.

Font, color and pattern properties are *inherited* from the object's owner if they are not set. This means that if you set the **textFont** of a stack, all the objects within that stack that do not have their **textFont** property set will use that text font.

The Message Path

When a message is sent to an object, it is often handled directly by a message handler in that object. However if no handler is present, the message will continue along a path until it finds a message handler

that can respond to it. This makes it possible to group similar functionality together at different levels within your application. This behavior applies both to event messages sent as a result of a user action, and custom messages sent by script. It is therefore possible to write libraries of common functions.

The object hierarchy is closely related to the *path* that a message travels on. In most cases, when an object passes a message on, the message goes to the object's owner in the object hierarchy.

The Message Path is detailed in the figure below.



Figure 48 – The Message Path

For example, suppose the user clicks a button in a main stack, causing LiveCode to send a **mouseUp** message to the button. If the button's script does not contain a handler for the **mouseUp** message, the message is passed along to the card the button is on. If the card's script contains a **mouseUp** handler, the handler is executed. But if the card does not handle the **mouseUp** message, it is passed on to the card's stack. If the stack script contains a **mouseUp** handler, the handler is executed. But if the stack does not handle the **mouseUp** message, it is passed on to the engine.

The engine is the end of the message path, and if a message reaches it, the engine takes any default action (e.g. inserting a character into a field or highlighting a button), then throws the message away.

If a message corresponding to a *custom* command or a *custom* function call reaches the end of the message path without finding a handler, instead of being thrown away, it causes an execution error.

Note: In order to be considered a background (as per the Message Path diagram above), a group must have its **backgroundBehavior** property is set to true.

If a stack's **dynamicPaths** property is set to true, message handlers in that stack use HyperCard's dynamic path behavior: if a handler uses the **go** or **find** command to go to a card other than the original card, that destination card's message path is inserted into the message path as long as the handler is on that card. The **dynamicPaths** property is provided for compatibility with imported HyperCard stacks, and is normally set to false, but you may encounter this behavior when working with a stack that was originally created in HyperCard.

The Message Target

The object that a message *was originally sent* to is called the message's target. You can get the target from within any handler in the message path by using the **target** function.

Tip: To get the name of the object whose script is currently executing, use the **me** keyword.

For example, if you click a button (causing a **mouseUp** message to be sent), and the button's script contains a **mouseUp** handler, then the **target** function returns the button's name. However, if the button doesn't handle the **mouseUp** message, it's passed to the card, and if the card has a **mouseUp** handler, it is executed in response to the message. In this case, the card's script is executing, but the **target** is not the card--it's the button that was originally clicked, because LiveCode sent the **mouseUp** message to the button.

Handlers with the Same Name

If two different objects in the message path each have a handler with the same name, the message is handled by the first object that receives it and has a handler for it.

For example, suppose that a button's script includes a **mouseUp** handler, and so does the stack script. If you click the button, a **mouseUp** message is sent to the button. Because the button's script has a **mouseUp** handler, the button handles the message, and it isn't passed any further. The message is

never sent to the stack script, so for this click event, the stack script's **mouseUp** handler is not executed.

Note: If an object's script has more than one handler with the same name, the first one is executed when the object receives the corresponding message. Other handlers in the same object's script with the same name are never executed.

Trapping Messages

When an object receives a message and a handler for that message is found, the handler is executed. Normally, a message that's been handled does not go any further along the message path. Its purpose having been served, it disappears. Such a message is said to have been *trapped* by the handler.

If you want to prevent a message from being passed further along the message path, but don't want to do anything with it, an empty handler for the message will block it from being passed on. This example prevents the **mouseDown** message from being acted on by the object the empty handler is in, as well as any objects below it in the object hierarchy:

```
on mouseDown
end mouseDown
```

You can use the same technique to block custom function calls, **setProp** triggers, and **getProp** calls.

Blocking System Messages

You can block system messages – for example those sent when you navigate to another card – from being sent while a handler is executing by setting the **lockMessages** property to true.

For example, if the handler opens another stack, LiveCode normally sends **openCard** and **openStack** messages to the stack. If the stack contains handlers for these messages that would cause unwanted behavior during this operation, you can use the **lockmessages** command before going to the stack in order to temporarily block these messages. When the handler finishes executing, the **lockMessages** property is automatically reset to its default value of false, and normal sending of messages resumes.

Tip: To block navigation messages while testing or debugging a stack, press Suppress Messages on the toolbar or choose **Development -> Suppress Messages**.

Passing a Message to the Next Object

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

```
on openCard
  doSpecialSetupForThisCard
  pass openCard -- let stack get the message too
end openCard
```

Selectively Trapping or Passing Messages

Some built-in messages, such as **keyDown**, trigger an action, so trapping the message prevents the action from being performed at all. The following example passes the **keyDown** message only if the character typed is a number:

```
on keyDown theKey
    if theKey is a number then pass keyDown
end keyDown
```

If you place this handler in a field's script, it allows the user to enter numbers, but any other keystrokes are trapped by the **keyDown** handler and not allowed to pass.

A similar principle applies to **setProp** triggers. If a **setProp** handler does not pass the **setProp** trigger, the custom property is not set.

Groups, Backgrounds & The Message Path

As you can see from the diagram above a group's position in the message path depends on whether the "Behave as Background" check box has been set (by script or using the **backgroundBehavior** property).

If background behavior is false, the group is in the message path for all controls it owns, but is not in the message path of any other object.

If the background behavior is true, the group is *also* in the message path for any cards it is placed on. If you send a message to a card control, the message passes through the control, then the card, then any background groups on the card in order of **number**, then the stack.

Since a group owns any controls that are part of the group, if you send a message to a control within a group, the group is in the message path for its own controls, regardless of whether it's background behavior is true or false. If a group has already received a message because it was originally sent to one of the controls in the group, the message is not sent through the group again after the card has handled it.

Tip: If you want a handler in a group's script to affect only the objects in the group, place the following statement at the beginning of the handler:

```
if the owner of the target is not me then pass message.
```

This filters out any objects that are not part of the group.

Commands and Functions

You use commands and functions to perform the actions of your application. Commands instruct the application to do something – such as play a movie, display a window, or change a property. Functions compute a value – different functions might add a column of numbers, or get the fifteenth line of a certain file, or find out whether a key is being pressed.

Using Built-in Commands and functions

LiveCode has over one hundred and fifty built-in commands, and over two hundred built-in functions, all of which are documented in the LiveCode Dictionary.

Commands

A command is an instruction to LiveCode to do something. A command is placed at the start of a statement (either the start of a line or after a control structure such as "then"). The command is followed by any parameters that specify the details of what the command is to do.

Here are some examples of how built-in commands are used in statements:

```
go next card -- go command
beep --beep command
set the hilite of me to true-- set command
```

Functions

A function call is a request to LiveCode for information. A function is stated using the name of the function, followed by opening and closing brackets that may contain any parameters that specify the details of what the function is to act on.

When you use a function in a statement, LiveCode calls the function to compute the specified information, then substitutes that information in the script as if the script had originally written that information in place of the function call. The information returned can therefore be *put* into a *variable* or other *container* by using the put command in front of a function.

Here's an example of how a function is used:

```
put round(22.3) into field "Number"
```

When this statement is executed, LiveCode calls the **round** function. When you round off 22.3, the resulting number is 22, so the statement puts the number 22 into the field.

The number you're rounding off is placed in parentheses after the **round** function's name. This number is called the function's parameter. A function can have one parameter, or none, or several. The parameters are enclosed in parentheses and, if there's more than one, they're separated with commas. Here are some examples:

```
put date() into myVariable -- date function, no parameters
put length("hello") into me -- length function, 1 parameter
get average(10,2,4) -- average function, 3 parameters
```

Important: A function call, by itself, is not a complete statement. You need to use a function call in conjunction with a command or control structure. (In the first example above, the **round** function is used with the **put** command.)

Writing function calls with the "the" form

If a built-in function has no parameters or one parameter, it can be written in a different form, with no parentheses:

```
put the date into myVariable -- date function
put the length of "hello" into me -- length function
```

If the function has no parameters, this form is written as the *functionName*. If it has one parameter, this form is written as the *functionName* of parameter.

The "the" form works the same way as the "()" form shown above, and you can use the two forms interchangeably for *built-in functions* with fewer than two parameters. The LiveCode Dictionary entry for each built-in function shows how to write both forms.

You can use the "the" form for built-in functions, but not for custom functions that you write. Writing custom functions is discussed later.

Custom Commands and Functions

You use custom commands and custom functions the same way as any other command or function.

Using custom commands

You can execute a custom command simply by typing the name of the command you want to send.

```
myCommand
```

You respond to a custom command's message in the same way, by writing a message handler with the name of the command:

```
on myCommand -- a custom command
  beep 3
  go next card
  add 1 to field "Card Number"
end myCommand
```

If you don't specify an object, the message is sent to the object whose script is being executed, and then passes up the message hierarchy as normal.

Like a built-in command, a custom command is an instruction to LiveCode to do something. You can include parameters with a custom command by passing them after the name:

```
checkForConnection "ftp://ftp.example.org"
makePrefsFile fileLoc,field "Preferences"
```

Built-in commands can have very flexible syntax:

```
go to card 3 of stack "Dialogs" as modal
group image "Help Icon" and field "Help Text"
hide button ID 22 with visual effect dissolve very fast
```

However, the syntax of custom commands is more limited. A custom command can have several parameters, and if there is more than one, they are separated by commas:

```
libURLDownloadToFile myFile,newPath,"downloadComplete"
revExecuteSQL myDatabaseID,field "Query","\*b" & "myvar"
```

But custom commands cannot use words like "and" and "to" to join parts of the command together, the way built-in commands can. Because of this, custom commands cannot be as English-like as built-in commands can be.

Using custom functions

When you use the custom function "fileHeader" in a statement, the function handler is executed, and the function call is replaced by the value in the **return** statement. This example shows how the function can be used:

```
put fileHeader(it) into myFileHeaderVar
```

And the corresponding custom function:

```
function fileHeader theFile
  put char 1 to 8 of URL ("file:" & theFile) into tempVar
  put binaryDecode("h*",tempVar) into tempVar
  return tempVar
end fileHeader
```

Like a built-in function or library function, a custom function call is a request for information. Here are a few examples showing the use of made-up custom functions:

```
get formattedPath("/Disk/Folder/File.txt")
put summaryText(field "Input") into field "Summary"
if handlerNames(myScript, "getProp") is empty then beep
```

Custom functions don't have a "the" form, and are always written with parentheses following the function name. If the function has parameters, they are placed inside the parentheses, separated by commas. If the function doesn't have parameters, the parentheses are empty.

Passing Parameters

A value that you pass from one handler to another is called a parameter.

In the example below, the following statement sends the "alertUser" message with a parameter:

```
alertUser "You clicked a button!"
```

The parameter "You clicked a button!" is passed to the "alertUser" handler, which accepts the parameter and places it in a parameter variable called "theMessage". The "alertUser" handler can then use the parameter in its statements:

```
on alertUser theMessage
  beep
  answer theMessage -- uses the parameter "theMessage"
end alertUser
```

Passing Multiple Parameters

If a statement passes more than one parameter, the parameters are separated by commas. The following example has two parameters, "theMessage" and "numberOfBeeps":

```
on seriouslyBugUser theMessage,numberOfBeeps
  beep numberOfBeeps
  answer theMessage
end seriouslyBugUser
```

To use this custom command, you call it like this:

```
seriouslyBugUser "Hello",5
```

When the "seriouslyBugUser" handler is executed with the statement above, the *theMessage* parameter is "Hello", and the *numberOfBeeps* parameter is 5.

Parameter Variables

In the example above, "theMessage" and "numberOfBeeps" are the parameter variables. You declare parameter variables in the first line of a handler. When the handler begins executing, the values you passed are placed in the parameter variables. You can use parameter variables the same way as ordinary variables: you can put data into them, use them in expressions, and so on.

Parameter variables are local variables, so they go out of existence as soon as the handler stops executing.

Parameter variable names

You can give a parameter any name that's a legal variable name. The names of variables must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore.

It is not the name, but the order of parameters that is significant.

Empty parameters

A handler can have any number of parameter variables regardless of the number of parameters passed to it. If there are more parameter variables than there are values to put in them, the remaining parameter variables are left empty. Consider the following handler, which has three parameter variables:

```
on processOrder itemName,itemSize,numberOfItems
  put itemName into field "Name"
  put itemSize into field "Size"
  if numberOfItems is empty then put 1 into field "Number"
  else put numberOfItems into field "Number"
end processOrder
```

The following statement places an order for one sweater:

```
processOrder "sweater","large"
```

The statement only passes two parameters, while the "processOrder" handler has three parameter variables, so the third parameter variable, "numberOfItems", is empty. Because the handler provides for the possibility that "numberOfItems" is empty, you can pass either two or three parameters to this handler.

Setting a default value for a parameter

To use a default value for a parameter, you check whether the parameter is empty. If it is, then no value has been passed, and you can simply put the desired default into the parameter, as in the following example:

```
logData theData,theFilePath
  if theFilePath is empty then
    put "logfile" into theFilePath
  end if
  put theData into URL ("file:" & theFilePath)
end logData
```

The "logData" handler puts data into a file, whose name and location you specify in the second parameter. If you only provide one parameter, the handler uses the filename "logfile" as the default value, and logs the data to that file:

```
logData field 1,"/Disk/Folder/data.txt" *-- specifies a file*
logData myVariable-- doesn't specify a file, uses "logfile"
```

The first statement above specifies the second parameter, so it doesn't use the default value. The second statement only specifies one parameter, so the data will be placed in "logfile" by default.

Implicit Parameters

If a statement passes more parameters than the receiving handler has parameter variables to hold them, the receiving handler can access the extra parameters with the **param** function:

```
function product firstFactor,secondFactor
  put firstFactor \* secondFactor into theProduct

  if the paramCount > 2 then

    repeat with x = 3 to the paramCount
      multiply theProduct by param(x)
    end repeat
  end if
  return theProduct
end product
```


The function above assumes that two parameters will be passed to be multiplied, but can multiply more numbers by using the **param** function to access parameters beyond the first two. The following statement uses the "product" custom function above to multiply four numbers together:

```
answer product(22,10,3,7)
```

When the "product" handler executes, the first two parameters – 22 and 10 – are placed in the parameter variables "firstFactor" and "secondFactor". The third parameter, 3, is accessed with the expression `param(3)`, and the fourth parameter, 7, is accessed with the expression `param(4)`.

Passing Parameters by Reference

Normally, if you pass a variable name as a parameter, that variable is not changed by anything the called handler does. This is because the variable itself is not passed, only its contents. Passing parameters in this way is called "passing by value" because the variable's value--not the variable itself--is what is passed.

If the name of a parameter variable is preceded with the `@` character, that parameter's value is a variable name, rather than the value in the variable. Changing the parameter variable in the called handler changes the value of the variable in the calling handler. This way of passing parameters is called "passing by reference", because you pass a reference to the variable itself instead of just its value.

For example, the following handler takes a parameter and adds 1 to it:

```
on setVariable @incomingVar
    add 1 to incomingVar
end setVariable
```

The following handler calls the "setVariable" handler above:

```
on mouseUp
    put 8 into someVariable
    setVariable someVariable*-- call by reference*
    answer "someVariable is now:" && someVariable
end mouseUp
```

Executing this **mouseUp** handler displays a dialog box that says "someVariable is now: 9". This is because, since "someVariable" was passed by reference to the "setVariable" handler, its value was changed when "setVariable" added 1 to the corresponding parameter variable.

You can pass parameters by reference to any custom function or custom command, simply by preceding the parameter name with the `@` character in the first line of the handler, as in the "setVariable" example handler above. Do not use the `@` character when referring to the parameter elsewhere in the handler.

Note: If a parameter is passed by reference, you can pass only variable names for that parameter. You cannot pass string literals or expressions using other containers such as fields. Trying to use the "setVariable" command described above using the following parameters will cause an execution error:

Empty parameters

If a handler defines a parameter as being passed by reference, you must include that parameter when calling the handler. Omitting it will cause an execution error.

Returning Values

Once a function handler has calculated a value, it needs a way to send the result back to the handler that called the function. And if an error occurs during a message handler, it needs a way to send an error message back to the calling handler.

The **return** control structure is used within a function handler to pass the resulting value back to the calling handler. The returned value is substituted for the function call in the calling statement, just like the value of a built-in function. Take another look at the example from above:

```
function expanded theString
  repeat for each character nextChar in theString
    put nextChar & space after expandedString
  end repeat
  return char 1 to -2 of expandedString
end expanded
```

In the custom function example above, the **return** control structure sends the spaced-out string back to the **mouseUp** handler that called the "expanded" function.

Note: The **return** control structure stops the handler, so it's usually the last line in the handler.

Returning an Error from a Message Handler

When used in a message handler, the **return** control structure serves a slightly different purpose: it returns an error message to the calling handler.

When used in a message handler, the **return** control structure sets the **result** function for the calling handler. If you want to return an error message to the calling handler, use the **return** control structure within the message handler.

Here's an example of a message handler that displays a dialog box:

```
on echoAMessage
  ask "What do you want to show?"
  if it is empty then return "No message!"
  else answer it
end echoAMessage
```

This handler asks the user to enter a message, then displays that message in a dialog box. If the user doesn't enter anything (or clicks Cancel), the handler sends an error message back to the calling handler. A handler that uses the "echoAMessage" custom command can check the **result** function to see whether the command successfully displayed a message:

```
on mouseUp
  echoAMessage
  if the result is empty then beep
end mouseUp
```

The **result** function is also set by many built-in commands in case of an error. If you check the **result** in a handler, the value belongs to whatever command – built-in or custom – that set it last, so if you're going to check the **result**, be sure to do so right after the command whose success you want to check.

Summary

In this topic, you have learned that:

A command instructs the application to do something, while a function requests the application to compute a value.

You create a custom command or custom function by writing a handler for it.

Values that you pass to a handler are called parameters.

To pass a parameter by reference, you precede its name with an @ sign in the first line of the handler.

When used in a function handler, the **return** control structure returns a value.

When used in a message handler, the **return** control structure returns an error message that can be accessed with the **result** function.

Variables

A variable is a place to store data that you create, which has no on-screen representation. Variables can hold any data you want to put into them. One way to think of a variable is as a box with a name on it. You can put anything you want into the box, and take it out later, by simply providing the variable's name:

```
put 1 into thisThing -- a variable named "thisThing"
put thisThing into field ID 234
put "Hello Again!" into line 2 of thisThing
```

But unlike some other types of *containers*, variables are non-permanent and aren't saved with the stack. Instead, variables are automatically deleted either when their handler is finished running or when you quit the application (depending on the variable's *scope*). You can also use the **delete variable** command to delete a variable. When a variable is deleted, not only the content of the variable disappears, but also the variable itself--the "box".

Tip: To save a variable's value, set a custom property of the stack to the value of the variable in your applications **closeStackRequest** or **shutDown** handler. To restore the variable, put the custom property into a variable in the application's **startUp** or **openStack** handler.

Variable Scope

The *scope* of a variable is the part of the application where the variable can be used. If you refer to a variable in a handler that's outside the variable's scope, you'll get either an execution error or an unexpected result.

There are three levels of variable scope: **local**, **scriptlocal**, and **global**. Every variable is one of these three types. The difference between these scopes is in where they can be used and how long their value lasts.

Local Variables

A local variable can be used only in the handler that creates it. Once the handler finishes executing, the variable is deleted. The next time you execute the handler, the variable starts from scratch: it does not retain what you put into it the last time the handler was executed.

To create a local variable, you simply put something into it. If you use the **put** command with a variable name that does not yet exist, the variable is automatically created as a local variable:

```
put true into myNewVar -- creates variable named "myNewVar"
```

Tip: While you can use almost any word that isn't a LiveCode language word – also known as a *reserved word* – for a variable name, it will help you greatly if you get into the habit of naming variables logically and consistently. For details of what names are allowed for variables, see the section on *Variable Names* below. For some recommendations on naming variables, see the section on *Tips for Writing Good Code*, below.

Alternatively, you can create a local variable explicitly by declaring it using the **local** command inside a handler:

Important: If you use a local variable in one handler, and that handler calls another handler, you can't use the local variable in the second handler. If you use a variable with the same name, LiveCode creates a second variable that is local to the second handler. But the two local variables don't affect each other, because they're in different handlers.

```
local myNewVar -- creates variable named "myNewVar"
put true into myNewVar -- puts a value into "myNewVar".
```

One common source of bugs involves misspelling a local variable name. Normally, doing so doesn't produce an execution error, because if you use a variable that doesn't exist, LiveCode creates it automatically. This means that if you misspell a variable name, LiveCode creates a new variable with the misspelled name. Such a bug may be difficult to track down because it can result in a variable having the wrong value without causing an error message.

To prevent this problem, you can require all local variables to be declared with the **local** command. You do this by turning on **Script -> Variable Checking** in the Code Editor menu bar. If this option is on, trying to use a local variable that doesn't exist will cause an execution error, instead of automatically creating it. Any misspelled variable names will therefore cause an obvious execution error when their handler is executed, making them easy to find.

Local variables are deleted when the handler that they're used in finishes executing. You can also use the **delete variable** command to delete a local variable.

Script Local Variables

A script local variable can be used in *any handler* in an object's script. You cannot use a script local variable in handlers in other objects' scripts. Unlike a local variable, a script local variable retains its value even after a handler finishes executing.

To create a script local variable, you must use the **local** command in the script, but outside any handler.

We recommend you always declare script local variables at the top of a script so they in one place and easy to find:

```
local mySharedVariable
on mouseDown
    put 2 into mySharedVariable
end mouseDown

on mouseUp
    answer mySharedVariable *-- displays "2"*
end mouseUp
```

Note: If you put the **local** command in a handler, instead of outside any handler, it creates a local variable instead. The command creates a script local variable only if you put it in the script but not within a handler.

You can also use the **delete variable** command to delete a script local variable.

Global Variables

A global variable can be used in any handler, *anywhere* in the application. Unlike a *local* variable, a global variable retains its value even after the handler that created it finishes executing. Unlike a *script local* variable, a global variable can be used by any handler in any object's script.

The same global variable can be used by any stack during a session. You can declare a global variable in one stack, and use it in others.

To create a global variable, you must declare it using the **global** command:

```
global someGlobalSetting
```

You must also use the **global** command at the start of a handler to make an existing global variable *available* to a handler. While a global variable can be used by any handler, you must do this in any handler you are going to use it in. If you don't declare a global variable before using it, the handler will not take the global variable's existence into account, and will simply create a local variable with the same name.

You can use the **global** command either inside a handler, or outside any handler at the top of a script (like a script local). If you use the command in a handler, the global variable can be used by any statement in that handler. If you use the command in a script but outside any handler, the global variable is available to every handler in that script.

The following example shows the use of a global variable in a button script. In this example, the variable is declared outside any handler, so the individual handlers don't need to declare it again:

```

global myGlobal -- declares global for the whole script
on mouseDown -- can use "myGlobal"
    put 1 into myGlobal
end mouseDown

on mouseUp -- can use "myGlobal"
    add 2 to myGlobal
    answer myGlobal-- displays "3"
end mouseUp

```

To use the same global variable in a handler where the variable isn't declared in the script, you must place the **global** declaration in the handler:

```

on mouseUp -- in another button's script
    global myGlobal
    add 5 to myGlobal
    answer myGlobal
end mouseUp

```

If you click the first button, then the second, the second button displays the number 8.

As with script local variables, we recommend you place all **global** declarations in scripts at the top of the script, making the declarations easy for you to find later.

Tip: You can get a list of existing global variables with the **globalNames** function. You can also choose **Development > Variable Watcher** to see a list of global variables and change their values. Or you can get the value using the Message Box.

Global variables are automatically deleted when you quit the application. You can also use the **delete variable** command to delete a global variable.

Variable Names

The names of variables must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore. You cannot use any LiveCode language word as a name of a variable.

Here are some examples of **legal** variable names:

```

someVariable
picture3
my\_new\_file
\_output

```

Here are some names that **cannot** be used as variable names:

```

3rdRock -- illegal as starts with a digit
this&That -- illegal as "&" cannot be used
My Variable -- illegal as it is more than one word

```

Avoid giving a variable the same name as a custom property. If you refer to a custom property, and there is a variable by the same name, LiveCode uses the contents of the variable as the name of the custom property. Generally this will produce unexpected results.

Global variables whose names begin with "gRev" are reserved by the LiveCode development environment.

See the section on *Tips for Writing Good Code* below for advice on choosing variable names.

Special Variable Types

Most of the time, you use variables that you create yourself, using the **local** or **global** commands, or simply by putting a value into a new variable in order to create it.

LiveCode also creates certain types of variables automatically: *parameter* variables, *command-line* variables, *environment* variables, and the special variable **it**.

Parameter variables

In a handler for a custom command or custom function, you can define parameters on the first line of the handler. For example, the following handler defines two parameters named "thisThing" and "thatThing":

```
on myHandler thisThing,thatThing
  add thisThing to thatThing
  subtract thatThing from field 2
end myHandler
```

When you use the custom command or custom function, you can pass values to it using the parameters:

```
myHandler 15,4+1
-- puts "15" into the parameter "thisThing",
-- and puts "5" into the parameter "thatThing"
```

When named parameters are used in a handler, they are called parameter variables. Within the handler, the parameters can be used in the same way as local variables: you can get their value, use them in expressions, and put data into them.

Like local variables, parameter variables persist only as long as the handler is executing.

Environment variables

Most operating systems that LiveCode supports provide information about the operating environment in *environment* variables.

You can access environment variables by prepending the **\$** character to the variable's name. For example, the following statement gets the contents of the LOGNAME environment variable, which holds the current user's login name:

```
get $LOGNAME
```

See your operating system's technical documentation to find out what environment variables are available.

You can also create your own environment variables by prepending the **\$** character to the new environment variable's name:

```
put field 3 into $MYENVVAR
```

Note: Environment variables behave like global variables and can be used in any handler. However, you do not need to use the **global** command to declare them before using them.

The environment variables that you create this way are available to the application, and are also exported to processes started up by the **shell** function or the **open process** command.

Command-line argument variables

If you start up the application from a command line, the command name is stored in the variable **\$0** and any arguments passed on the command line are stored in numbered variables starting with the **\$** character.

For example, if you start the application by typing the following shell command:

```
myrevapp -h name
```

then the variable **\$0** contains "myrevapp" (the name of the application), **\$1** contains "-h", and **\$2** contains "name".

Note: Command-line argument variables behave like global variables and can be used in any handler. However, you do not need to use the **global** command to declare them before using them.

The special variable "it"

The **it** variable is a special local variable used by LiveCode to store certain results.

Certain commands--such as **get**, **convert**, **read from file**, **ask**, and **answer**--put their results in this special variable. For a complete list of commands that use the **it** variable, see the entry for **it** in the LiveCode Dictionary. The following example shows how the **answer** command uses the **it** variable:

```
on mouseUp
  answer "Go where?" with "Backward" or "Forward"
  -- the answer command puts the button the user clicked
  -- into the it variable:
  if it is "Backward" then
    go back
  else
    go next
  end if
end mouseUp
```

You can use the **it** variable in the same way as any other local variable, using the **put** command to put

things into it and using the variable in expressions to work with the contents.

Array Variables

A variable can hold more than a single value. A variable that holds more than one value is called an array, and each of the values it holds is called an element. Each element has its own name (called the element's key).

If you think of a variable as a box with a name, you can think of an array as a box with compartments inside it. Each compartment is an *element*, and each compartment has a name, or *key*, of its own.

You specify an element of an array variable by using the variable name along with the element's key. You enclose the key in square brackets. The key may be a name, number or variable. Here's an example that shows how to put data into one element of an array:

```
put "ABC" into myVariable["myKeyName"]
```

Note: If you use a key that's not a number or variable, you should enclose the key's name in double quotes whenever you refer to it. This prevents problems in case there is a variable or reserved word with the same name.

You can use any element of an array variable in the same way you use the whole variable: put data into the element (or before or after it) and find out what data it contains. In this way, any element of a variable is like a variable in and of itself.

Array elements may contain nested or sub-elements, making them multi-dimensional. This type of array is ideal for processing hierarchical data structures such as trees or XML. To access a sub-element, simply declare it using an additional set of square brackets.

```
put "ABC" into myVariable["myKeyName"]["aChildElement"]
```

You may nest elements within themselves to any number of levels.

Deleting Elements of an Array

You use the **delete variable** command to remove one element from an array variable, in the same way you delete a variable. To delete an element, you specify both the variable and the element's key:

```
delete variable myVar["myElement"] delete variable  
myVar["myElement"]["child1"]
```

This statement removes the element named "myElement" from the variable "myVar", but does not delete the other elements in the array.

Listing the Elements in an Array

Tip: To delete the contents of an element without deleting the element itself, put **empty** into the element:

```
put empty into myVar["myElement"]
```

You use the **keys** function to list the elements in an array variable. The **keys** function returns a list of elements, one per line:

```
put the keys of myArray into listOfElements
```

Listing Nested Elements within an Element

You use the **keys** function to list the child elements of an element within an array variable. The **keys** function returns a list of elements, one per line:

```
put the keys of myArray["node25"] into listOfElements
```

Transforming a List of Data into an Array

The **split** command separates the parts of a variable into elements of an array. You can specify what character in the list you want the data to be split by. The data will be converted into a set of elements named according to where it is split. For example:

```
put "A apple,B bottle,C cradle" into myVariable
split myVariable by comma and space
```

Will result in the following:

KEY	VALUE
A	apple
B	bottle
C	cradle

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

Combining the Elements of an Array into a List

The **combine** command combines the elements of the *array* into a single variable. After the command is finished executing, the variable specified by *array* is no longer an array.

For example:

```
combine myArray using return
```

Will combine the contents of the each element of the original *array* so that they appear on a separate line.

For more information, see the **combine** command in the LiveCode Dictionary.

Nesting an Array

You may place an entire array as a child of an element by putting an array variable into an element of

another array. For example:

```
put tMyArray into tBigArray["node50"]
```

Will result in the entire array being placed as a child of `node50` within `tBigArray`.

More Information

For more information about using array variables, see the guide on *Processing Text and Data*.

Constants

A constant is a value that has a name. Like a variable, a constant is set in your script. Unlike variables, constants cannot be changed.

When you use a constant, LiveCode substitutes the value of the constant for its name. The following example uses a constant named "slash":

```
put slash after field "Expressions" *-- displays "/"
```

You create a new constant using the **constant** command.

You cannot put anything into a constant once it's been created.

Built-in constants

The LiveCode language defines several constants, such as **return**, **space**, and **comma**, for characters that have special meaning in scripts and therefore can't be entered literally into an expression.

Tip: To see a list of all built-in constants, open the Documentation window, click LiveCode Dictionary, and choose "Constants" from the menu at the top of the window.

User-defined constants

You can also define your own constants using the **constant** command:

```
constant myName="Joe Smith"
```

Like variables, constants can have different scope depending on how they are created. A constant can be defined as either a local constant or a script local constant:

If you place the **constant** command in a handler, the constant can be used only in that handler.

If you place the **constant** command in a script, but outside any handler, the constant can be used in any handler in the script.

Containers, Operators & Sources of Value

What is a Container?

Containers are sources of information that can be edited using chunk expressions. In addition to variables, LiveCode has six other container types: fields, buttons, images, URLs, the selection, and the message box.

Fields, buttons, and imported images are all LiveCode objects. All display their content on the screen, in different ways, and the contents of all three are saved when you save the stack they are in. URLs refer to external resources (either files on the system, or items on an Internet server). The Message Box is a special container that's part of the development environment. All of these containers are covered in more detail in their respective sections.

Setting and Retrieving Data from Containers

You use the **put** command to place data into a container, or to place data from a container into another container or variable. Containers support the use of chunk expressions, the ability to specify a portion of a container by referring to it in English. For more details, see the section on *Chunk Expressions*.

What are Sources of Value?

Sources of value are like containers. They can be retrieved using the **get** command. However unlike containers, they cannot be set using the **put** command. Sources of value include properties, function calls, literal strings, and constants.

Getting and Setting Properties

You can use the **get** command to retrieve data from properties. Properties may be set using the **set** command. When retrieving data you may use chunk expressions. However you must set a property in its entirety, using the set command. Properties are covered in more detail in the section on *Properties*. Examples of retrieving and setting properties are to be found throughout this *User's Guide*.

What are Literal Strings?

A literal string is a string of characters whose value is itself. If the string is a number, the value is that number.

Using Literal Strings

When you use a literal string in an expression, LiveCode simply substitutes the string for itself:

```
put "Hello World!" into field 1
get 1 + 2 + it
put 1 - 4.234 into field "Result"
```

Quoting strings

Literal strings that consist of more than one word or are reserved words in the LiveCode language must be enclosed in double quotes:

```
put "This is a test" into myVar -- works
put This is a test into myVar -- **DOESN'T WORK - not quoted**
put That into myVar -- works
put This into myVar -- **DOESN'T WORK - reserved word**
```

In some contexts, you can use an unquoted one-word literal string without causing a script error. However, you should make a practice of always quoting literal strings (other than numbers), because it ensures that the statement will continue to work properly even if the string becomes a reserved LiveCode word in the future.

If the **Script -> Variable Checking** option is set to true, compiling a script that contains an unquoted literal string causes a script error.

What are Operators?

Use operators to put together, compare or perform an operation on data. Use a *String Operator* to combine data. Use a *Numeric Operator* to perform a calculation. Finally, use a *Logical Operator* to return true or false.

Numeric Operators

Numeric operators produce a number as their result. Numeric operators include the arithmetic operators (+, -, *, /, **mod**, **div**, and ^) and the bitwise operators (**bitAnd**, **bitOr**, **bitXor**, and **bitNot**). For individual usage instructions, look up the operator you wish to use in the LiveCode Dictionary.

For example:

```
put "1+2 =" && 1+2 into field "Eq" -- displays "1+2 = 3"
```

String operators

String operators produce a string of characters as their result. String operators are the concatenation operators (&, &&, and ,).

```
put "1+2 =" && 1+2 into field "Eq" -- displays "1+2 = 3"
```

Logical Operators

Logical operators produce either "true" or "false" as their result.

Logical operators include the comparison operators (=, \<>, \<, >, \<=, >=), existence operators (**there is a**, **there is no**, **is in**, **is not in**, **is among**, **is not among**, **contains**), data type operators (**is a**, **is not a**), geometry operators (**is within**, **is not within**), and basic logical operators (**and**, **or**, **not**).

```
if the platform is "MacOS" and field "Time" < zero then ...
```

Binary versus Unary Operators

Operators can use either one argument (a unary operator) or two arguments (a binary operator):

```
"a" & "b" -- & is binary
there is a field "c" -- **there is a** is unary
```

The **bitNot**, **there is a**, **there is no**, **is a**, **is not a**, and **not** operators are unary operators. All other operators are binary operators.

Conversion of Values

LiveCode converts values in expressions to whatever type of data is needed for the operation. This conversion happens automatically, so you don't need to know what type of data you are dealing with in advance (something that in other languages is called "type casting").

For example, suppose you have the following statement:

```
put char 2 of "123" + char 3 of "456" into field 3
```

Character 2 of the literal string "123" is the single-character string "2", and character 3 of the literal string "456" is the single-character string "6". When LiveCode uses the **+** operator, it automatically converts these strings to numbers so that they can be added together. Then it converts the resulting number back to a string so that it can be placed in a field as text.

Operator Precedence

When you combine sources of value using operators, LiveCode evaluates each source of value in the expression. Next, it applies any operators to arrive at a final value for the expression. LiveCode does not necessarily apply all the operators in right-to-left order.

Instead, it uses the operator precedence order to determine how to compute expressions that include more than one operator. Precedence determines the order in which LiveCode carries out calculations in expressions. If an expression contains more than one operator, the operators with higher precedence are calculated before operators with lower precedence.

Figure 49 – Operator Precedence

Grouping	()	Any part of the expression in parentheses is evaluated first. If parentheses are nested, the innermost values are evaluated first.
Unary	- not bitNot there is a there is no	Next, unary operations (that act on only one operand) are done. This includes unary minus (which makes a number negative).
Exponentiation	^ *	Next, exponentiation operations are done.
Multiplication	/ div mod	Multiplication and division are done next. These are numeric operators and result in a number.
Addition	+ -	Addition and subtraction are done next. These are numeric operators and result in a number.
Concatenation	& && ,	Operations that join two strings are done next. These are string operators and result in a string.

	\<	
	\<=	
	>	
	>=	
	contains	
	is among	
	is not among	
	is in	
	is not in	
	is within	
	is not within	
	is a	
	is not a	
	=, is	
Comparison	\<>, !=, ?, is not	Operations that compare two values are done next. These are logical operators and result in either true or false.
Equality		Operations that compare two values for equality are done next. These are logical operators and result in either true or false.
bitAnd	bitAnd	bitAnd operations are done next.
bitXOr	bitXOr	bitXOr operations are done next.
bitOr	bitOr	bitOr operations are done next.
and	and	and operations are done next.
or	or	or operations are done last.
function calls		Functions are evaluated after all possible operators in the function's parameters are evaluated.

Using the Grouping Operator ()

Suppose you want to change the precedence used in an expression. The grouping operator () has higher precedence than all other operators, so you can use it to change the order in which operators are evaluated. In other words, if you enclose part of an expression in parentheses, any operations inside the parentheses are evaluated before anything else happens.

For example, function calls have higher precedence than division, so the expression the sin of 1/4 means "get the sine of 1, then divide it by 4":

```
get the sin of 1/4 -- does the sin function first
```

If you want the sine of 1/4, you need to change the order of evaluation so that the division is done first. You do this by surrounding the part you want evaluated first with parentheses:

```
get the sin of (1/4) -- does the division first
```

If parentheses are nested, the expression within the innermost set of parentheses is evaluated first.

Factors and Expressions

An expression is any source of value, or combination of sources of value. Any of the sources of value discussed above – containers, properties, function calls, literal strings, and constants – are simple

expressions. You use operators to combine sources of value to produce more complex expressions.

Defining Factors

A factor is the first fully resolvable portion of an expression. (All factors are expressions, but not all expressions are factors.) A factor can be either a source of value, or an expression that combines sources of value but doesn't include any binary operators outside parentheses. Enclosing an expression in parentheses turns it into a factor.

These examples show some expressions, along with the first factor in each expression, to help you see the distinction:

Expression	First factor
3 + 4	3
(3 + 4)	(3 + 4)
(3 + 4)/field 4	(3 + 4)
field 6 * pi	field 6
sin of pi/4	sin of pi
sin(pi/4)	sin(pi/4)
sin(pi/4) * exp2(12)	sin(pi/4)
whole world	whole
"whole world"	"whole world"

When it matters

The distinction between factors and expressions matters when you're using the **the** form of built-in functions, when you use **URL**, and when you refer to objects.

If you use the **the** form of a built-in function that has a parameter, the parameter must be a factor, not an expression:

```
get the sqrt of 4 + 5 -- yields 7
get sqrt(4+5) -- yields 3
get the sqrt of (4 + 5) -- yields 3
```

In the first example above, although we intended to get the square root of 9, we actually ended up with the square root of 4. This is because the expression `4 + 5` is not a factor (because it contains a binary operator that's not inside parentheses). The first factor in the expression `4 + 5` is 4, so the first example gets the square root of 4 (which is 2), then adds 5 to that result.

The second example avoids this problem because it doesn't use the "the" form of the function. Since the parameter is enclosed in parentheses, you can use either a factor or an expression, and obtain the intended result, the square root of 9.

In the third example, we turn the expression `4 + 5` into a factor by surrounding it with parentheses. Since the parameter is now a factor, we can get the correct result, even using the **the** form of the function.

When referring to URLs, the URL is a factor:


```
get URL "file:myfile.txt" -- works
get URL "file:" & "myfile.txt" -- **DOESN'T WORK**
get URL ("file:" & "myfile.txt") -- works
```

In the first example, the URL specified is a factor because it is a simple string with no operators.

The **URL** in the second example is not a factor, because it includes the binary operator **&**, so the **get** command tries to get the URL "file:" – which is nonexistent--and concatenate the content of that **URL** with the string "myfile.txt".

In the third example, we turn the **URL** into a factor by surrounding it with parentheses, providing the expected result.

When referring to **cards** or **backgrounds**, the **name**, **number** or **ID** of the object is an expression:

```
go card 1 + 1 -- goes to card 2
go card 3 of background "Data" && "Cards"
-- goes to first card with the group "Data Cards"
```

However, when referring to controls (including groups) or stacks, the **name**, **number**, or **ID** of the object is a factor:

```
answer field 1 + 10 -- displays field 1 content + 10
answer field (1 + 10) -- displays field 11 content
select button "My" && "Button" -- **DOESN'T WORK**
select button ("My" && "Button") -- works
```

Making Decisions

You make decisions using the **if ... then ... else** control structure or, if you want to choose from a list of options, use the **switch** control structure.

If ... then ... else

Use the **if** control structure to execute a statement or list of statements under certain circumstances. For example, you may want your application to respond differently depending on a user action.

The **if** control structure always begins with the word **if**. There are four forms of the **if** control structure:

```
if condition then statement [else elseStatement]
```

This form may have a line break before the words **then** or **else** or both.

```

if condition then
    statementList
[else
    elseStatementList]
end if

if condition
then statement
[else
    elseStatementList
end if]

if condition then
    statementList
else elseStatement

```

The *condition* is any expression that evaluates to true or false. The *statementList* or *elseStatementList* consists of one or more LiveCode statements, and can also include if, switch, try, or repeat control structures. The *statement* or *elseStatement* consists of a single LiveCode statement.

If the *condition* evaluates to true, the *statement* or *statementList* is executed; if the *condition* evaluates to false, the *statement* or *statementList* is skipped. If the if control structure contains an else clause, the *elseStatement* or *elseStatementList* is executed if the *condition* is false.

If one if control structure is nested inside another, use of the second form described above is recommended, since the other forms may cause ambiguities in interpreting which else clause belongs with which if statement.

The if control structure is most suitable when you want to check a single *condition*. If you need to check for multiple possibilities, doing something different for each one, use a switch control structure instead.

Switch

Use the **switch** control structure when you want to choose among several possible values for an expression and then execute a set of statements that depends on the value.

```

switch [switchExpression]
case {caseValue | caseCondition}
    [statementList]
[default defaultStatementList]
end switch

```

The **switch** control structure begins with the word switch on a single line, with an optional *switchExpression*. The **switch** line is followed by one or more case sections. Each case section begins with the case keyword, followed by either a *caseValue* (if a *switchExpression* was included on the **switch** line) or a *caseCondition* (if no *switchExpression* was included). If the *caseValue* is equal to the *switchExpression*, or the *caseCondition* evaluates to true, LiveCode begins executing the following statements.

The **case** sections may be followed by an optional default section. If no **break** statement has been encountered yet in the switch control structure, the statements in the default section are executed. The **switch** structure ends with an end switch statement.

The *switchExpression* is any expression. The *caseValue* is any expression. If the *caseValue* evaluates to the same value as the *switchExpression*, the condition is matched for that case section. The *caseCondition* is any expression that evaluates to true or false. (If the *caseCondition* evaluates to true, the condition is matched for that case section. Each *statementList* consists of one or more LiveCode statements, and can also include if, switch, try, or repeat control structures. The *defaultStatementList* consists of one or more LiveCode statements.

Flow of control in a **switch** structure is less complicated than it looks. In general, when LiveCode enters a **switch** control structure, it looks for the first case section whose *caseValue* is equal to the *switchExpression*, or whose *caseCondition* is true. When a matching condition is found, all statements following it are executed--even statements in another case section--until either a break statement is encountered or the **switch** control structure ends.

This means that if you do not end a **case** section's *statementList* with a break statement, the statements in all the following case sections (and the default section) are executed even if those case sections don't have a matching *caseValue* or a true *caseCondition*. Occasionally, this behavior is useful. However, in most cases, you should place a break statement at the end of each *statementList*. This ensures that only one *statementList* is executed, and the rest are skipped.

This also means that you can attach more than one *caseValue* or *caseCondition* to the same *statementList*, simply by placing one case line above the next. The following example beeps if the current card is either the last or first card, and goes to the next card otherwise:

```
switch (the number of this card)
case 1
case (the number of cards)
    -- either of the above conditions execute the following
    -- statement:
    beep
    break
default
    go next card
end switch
```

There is no limit to the number of **case** sections you can include in a **switch** control structure, although the more case sections there are, the more expressions LiveCode must evaluate and the more slowly the **switch** structure executes.

Extending the Message Path

This section deals with how to extend the message path, either by adding code libraries to the message path, or by sending messages directly to objects that are not currently in the message path.

Creating a Code Library

A library is a set of custom commands and custom functions for a specific application or a specific area of functionality. You can create a library for any purpose you want, and put any custom commands and functions into it that you need. Libraries are typically used to store routines that are common across your application. You can also exchange useful libraries with other developers.

To create a code library, place the handlers you want to use in any object that's available in your stack. This object is now a code library. Then use the **insert script** command to add that object to the message path. To insert the script of a stack into the message path, use the **start using** command. Typically you would run one of these commands as your application is starting up, so that all the scripts

can access the libraries you require. Libraries do not need to be in the same stack or even stack file, you can load any stack on disk and then load the libraries within it to make them available to all running stacks. This makes it easy to design your application in modules, share code with other developers or update your application libraries without modifying your application. You can design your standalone application to work in the same way, making it easy to update it using a small patch utility, i.e. without having to reinstall the entire application.

Using backScripts

To make the script of an object available to any other handler in LiveCode:

```
insert script of card "Library" into back
```

The script of an object that's been inserted into the back, as in the above example, is called a **backScript**. Such an object is placed last in the message path of all objects. It receives messages after any other object, and just before the engine receives the message. The **backScript** object remains in the message path until the end of the session (or until you remove it with the **remove script** command.)

Because a **backScript** receives messages after all other objects in the message path, if you place a handler in the **backScript**, it eventually gets all the corresponding messages, no matter what object sent them, unless another object handles them first.

Using frontScripts

You can also extend the message path by placing objects into the front of the message path, before any other object. The script of such an object is called a frontscript, and you use the **insert script** command to place the object in the front:

```
insert script of button "Interceptor" into front
```

Because the object is in the front of the message path, it receives messages even before the target of the message. For example, if you click a button, any objects in the front of the message path receive the **mouseUp** message first, before the button. If you place a handler in a frontscript, it receives all the corresponding messages before any other object can handle them.

Use a frontscript when you want to be able to handle a message even if the target object has a handler for it. For example, the LiveCode development environment displays a contextual menu when you **Control-Shift-right-click** an object. It does this with a **mouseDown** handler in a frontscript. Whenever you click an object, the frontscript receives the **mouseDown** message first, and checks whether the needed keys are being pressed. If they are, the handler displays the contextual menu; the **mouseDown** message is trapped and goes no further. Otherwise, the handler passes the message to the next object in the message path, which is the object you clicked.

Using a Stack's Script with *start using*

The **start using** command is similar to the **insert script** command, but can be used only to place stacks, not other object types, in the message path.

The **start using** command inserts the stack at the end of the message path, *after* the object's stack and main stack, but *before* objects that have been inserted into the back with **insert script**.

Sending Messages Directly to Objects

If the handler you want to use is not in the message path, instead of inserting it into the path to make it a code library, you can use the **send** command to directly **send** a message to the object whose script contains the handler.

Tip: You can use this technique to skip a portion of the message path, by sending the message directly to an object further up the hierarchy.

```
on mouseUp
  -- in button script
  send "mouseUp" to this stack
end mouseUp
```

For example, suppose you have a handler for a custom command called *myCommand* in the script of a button, and you want to execute that handler from a card script. Since the button is not in the card's message path, you can't simply use the command in the script, because if the *myCommand* message goes through the card's message path, it won't find the handler. The following statement sends the message directly to the button whose script includes the handler:

```
send "myCommand" to button "Stuff" of card "Stuff Card"
```

Important: When you send a message to an object, the message path for that message starts with the target object. For example, if a stack script contains a **send** command that sends a message to a button, the message moves through the button's message path, not the stack's.

If you want to use a custom function whose function handler is not in the message path, you can use the **value** function to specify the object. The **value** function can be thought of as an equivalent of the **send** command for function calls.

For example, if card 1 of a stack contains a function handler called "myFunction" you can use the function from within the script of card 3 with the following statement:

```
get value("myFunction(1,2,3)", card 1)
```

The **send** Command versus the **call** Command

The **call** command is similar to the **send** command. Like the **send** command, the **call** command sends a message to the specified object, allowing you to use handlers that aren't in the message path.

The difference between **send** and **call** is how they handle object references in the handler that is triggered by the sent message. If the message is sent by the **send** command, object references in the handler are treated relative to the object you sent the message to.

For example, suppose card 1 of a stack contains the following handler:

```
on showCard
  answer the number of this card
end showCard
```

If a handler in the script of card 3 uses the **send** command to send a "showCard" message to card 1, the dialog box displays "1", the number of the card the handler is on. However, if the same handler uses the **call** command instead, the dialog box displays "3", the number of the card that used the **call** command. In other words, handlers that are triggered by the **send** command use the context of the object the handler is in, while handlers that are triggered by the **call** command use the context of the object that triggered the handler.

Writing reusable code using behaviors

Behaviors are a method to create common functionality between objects without duplicating the scripts.

An object with a **behavior** set will act as though its script was set to the script of the **behavior** button. If multiple objects share the same behavior, each will have its own set of script local variables. Any references to **me**, the **owner of me**, etc will resolve to the child object currently executing.

Behavior scripts must be stored in button objects. To set an object's behavior, refer to the long id of the button containing the behavior script.

Timer Based Messaging

Timers allow you to schedule events to happen in the future. Use timers for updating the display at regular intervals, processing data in chunks, playing animations, displaying status bars, or anywhere where you need to schedule events. Messages can be scheduled with millisecond precision and fire many times a second to create an animation, or can be scheduled to arrive hours later. When you schedule an event to be delivered in the future, LiveCode continues to respond to user events as normal. This makes timer based messaging ideal where you want your user interface to remain responsive while doing data processing or updating the display.

Delivering a Message in the Future

To deliver a message after a specified time period, use the *in time* form of the send command.

```
send "updateStatus" to me in 20 seconds
send "updateAnimation" to me in 33 milliseconds
```

Repeating a Timer Message

If you want to send a message repeatedly, for example to continuously draw frames in an animation, simply send the same message again at the end of the message handler. The following example starts an animation when the button is clicked on, then updates the frame at 30 frames per second (every 33 milliseconds).

```
on mouseUp
    updateAnimation
end mouseUp

on updateAnimation
    -- insert code to update animation here
    send updateAnimation to me in 33 milliseconds
end updateAnimation
```

The result will be that the *updateAnimation* message is sent, it will update the screen then send itself to itself in a 33 milliseconds. The message will continue to be delivered indefinitely. Therefore it is important that you ensure there is a condition in the message handler that will exit when the task is done, without sending the message again. Alternatively you can directly cancel any message to stop the loop (see the section *Canceling a Timer Message* below for more details).

To create a *smooth script driven animation*, you should allow for the time it takes to redraw the display and for any interruptions (e.g. another intensive process causing your message to be delivered late). Thus it is best to send the message a little more frequently than you need to redraw the screen, and to check each time the message is activated what frame you should currently be on.

```
-- variables will be available to all script handlers
local sTotalFrames, sStart

on mouseUp
    -- store the current start time
    put the milliseconds into sStart
    -- put the number of frames in your animation here
    put 100 into sTotalFrames
    updateAnimation
end mouseUp

on updateAnimation
    -- calculate milliseconds since start
    put the milliseconds - sStart into tElapsedTime
    put round(tElapsedTime / 33) into tFrameNumber
    -- calculate current frame number
    if tFrameNumber > sTotalFrames then
        -- if we have reached the end
        set the currentFrame of image 1 to sTotalFrames
        -- draw the last frame
        exit updateAnimation
        -- exit without sending updateAnimation again
    end if

    -- insert code for drawing the screen here
    -- in this example we assume the animation is a simple
    -- animated GIF
    -- but you could have any type of operation here, e.g.
    -- setting the transform of a vector
    set the currentFrame of image 1 to tFrameNumber
    -- send another updateAnimation message
    send updateAnimation to me in 22 milliseconds
end updateAnimation
```

For more example scripts involving timers and animation, see the *Sample Scripts* section of the online documentation.

Canceling a Timer Message

When a timer message is sent, the **result** function contains the ID of the message generated. Use the **cancel** command to cancel that message, preventing it from being delivered. In the following example, a dialog will be displayed 20 seconds after the user moves the mouse over a button. If the mouse moves out of the button before the 20 seconds has elapsed, the dialog will not be displayed.

```

local LTimerID
on mouseEnter
    send "displayDialog" to me in 20 seconds
    put the result into LTimerID
end mouseEnter

on mouseLeave
    cancel LTimerID
end mouseLeave

on displayDialog
    answer "The mouse was over this button for 20 seconds."
end displayDialog

```

Important: Ensure you have a way to cancel any pending timer messages. Typically you may want to ensure that a timer is cancelled when the card is closed. For example a timer message that draws an animation on the current card will generate a script error if the card is changed and the message is still sent, as the script will no longer be able to find the objects.

Displaying a List of Pending Timer Messages

You can get a list of all the currently pending timer based messages using the **pendingMessages** function.

```

put the pendingMessages into tMessagesList

```

tMessagesList will now contain a list of messages, one per line. Each line consists of four items, separated by commas: *the message ID*, the time the message is scheduled for, *the message name*, the long ID property of the object that the message will be sent to

Tip: You can see a list of all pending messages using the *Message Box* pending messages tab – the fifth icon along. For more details, see the section on the *Message Box*.

For more details, see the *pendingMessages* entry in the *LiveCode Dictionary*.

Tip: To cancel all currently pending messages indiscriminately, use the following repeat loop:

```

repeat for each line l in the pendingMessages
    cancel (item 1 of l)
end repeat

```

Tips for Writing Good Code

It's worth taking time to establish some conventions in how you write code. There are many benefits to making your coding practices consistent. Consistently named variables are easier to debug during development because you will always know the scope of a particular variable (whether it's a global, or applies only to the current handler, etc.). It makes it easier to understand the code you have written when you come to read it 6 months later. By making appropriate use of functions and libraries, you make

your program more modular and therefore easier to modify and add features to in the future. A comprehensive guide to writing good code is beyond the scope of this manual. But we thought it would be helpful to give you some hints and tips

Variable naming

Use consistent variable names to make your code easier to understand. It may not seem important now, but when you have forgotten how it works 6 months later it will help to make it readable. It also makes it easier to exchange you code with other members of the LiveCode community, when you need to get help with something.

Character	Example	Usage
g	gVar	Global variables
t	tVar	Handler-local variables
s	sVar	Script-local variables
p	pVar	Parameters
k	kVar	Constants
c	cVar	Custom properties

Variable scope

As a general rule of thumb, use a variable with just enough scope and no more for the task at hand. In other words, if a handler local variable is all you need for a calculation, don't use a script local. If a script local would do, don't use a global. This makes it less likely that you will introduce unexpected errors into your code by using the same variable name for different purposes. A handler local variable only has meaning within that handler so you can safely use the same variable name in other handlers.

Use of comments

Comment your code as you go. Don't write comments where the meaning is obvious. But even a sentence next to a complex routine will help you or others to quickly understand it later.

Use of functions

If you're writing a lot of code, consider whether it can be written as a series of functions instead of as a single monolithic handler. This makes it much easier to understand the function of each discrete section of code. The more you can make these functions "black boxes", that take input and produce output without dependences, the easier it will be if you later want to change how that aspect of the code works.

Explicit Variables

If you're working on a larger code base you may want to turn on the Variable Checking option in the Script menu of the Code Editor. This makes LiveCode require that you declare all variables (even script locals) before you can compile your script. It also requires you to place all literal strings in quotes. This method of coding can help you to quickly track down errors as you go by making you think a little about what variables you are going to use at an early stage, and pointing out spelling mistakes.

What's most important is that you develop your own consistent style and then stick to it. Once you have been applying some of these techniques for a short time, they will become second nature.

For a more in-depth look at this subject, we recommend [Fourth World's Scripting Style Guide](#)

Programming a User Interface

Introduction

The user interface for your application is often one of its most important features. The *LiveCode Script* guide shows how to build a user interface using LiveCode's tools and development environment. In this guide we look at how you can edit, or even build a user interface programmatically. Everything you can do using the built-in tools you can also do programmatically. You can even create and modify a user interface at run time in a standalone application, or provide interactive methods for your users to modify specific aspects of your application. This set of capabilities allow you to produce applications that construct their interface using XML files or a custom data structure, programmatically construct aspects of complex interfaces, modify their look using user specified parameters, create themed or skinned interface options, build your own interface editing tools that plug-in to LiveCode's IDE and much more. You can also create custom objects and attach your own virtual behaviors and custom properties to them. We recommend you spend a little time becoming familiar with building an interface using the tools in the development environment before creating or editing an interface programmatically.

Referring to Objects

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

Referring to objects by name

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

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

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

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

Referring to objects by number

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

You can refer to an object using its object type followed by its number. For example, to refer to the third-from-the-back field on a card, use the phrase "field 3":

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

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

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

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

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

Referring to objects by ID

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

You can refer to an object using its object type, then keyword **ID**, followed by its **ID** number. For example, to refer to a card whose **ID** property is 1154, use the phrase card **ID** 1154:

```
go to card ID 1154
```

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

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

Referring to objects by ordinal

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

```
go to last card
```

The special descriptor 'this'

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

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

Control references

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

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

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

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

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

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

Nested Object References

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

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

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

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

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

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

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

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

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

called "Other Stack":

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

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

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

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

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

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

Avoid using numbers as object names

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

For example, create three fields on a new card, naming the first one "3", the second one "2", and the third one "1". In other words, the creation order of those three fields is the reverse of the numerical names you've given them. Now use the message box to:

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

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

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

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

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

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

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

You could also construct numerical field names:

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

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

Properties

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

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

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

Using Object Properties

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

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

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

Referring to properties

Property references consist of the word **the**, the property name, the word **of**, and a reference to the object:

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

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

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

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

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

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

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

Changing properties

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

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

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

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

Read-only properties

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

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

Changing a part of a property

Properties are not containers, so you cannot use a chunk expression to change a *part* of the property. However, you can use a chunk expression to *examine* part of a property. For example, you cannot set line 1 of a property to a new value: you must set the whole property. For more details see the section *Chunk Expressions* in the chapter on *Processing Text and Data*.

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

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

Custom properties and virtual properties

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

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

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

Property Inheritance

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

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

This process of first checking the object, then the object's owner, then the object that owns that object, and so on, is called *inheritance* of properties. Each object inherits the background color of the object above it in the hierarchy. Similar inheritance rules apply to the **foregroundColor**, **topColor**, **bottomColor**, **borderColor**, **shadowColor**, and **focusColor** properties, to their corresponding pattern properties, and to the **textFont**, **textSize**, and **textStyle** properties.

Overriding inheritance

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

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

The effective keyword

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

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

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

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

Global Properties

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

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

A few properties are *both* global *and* object properties. For example, the **paintCompression** is a global property, and also a property of images. For these properties, the global setting is separate from the setting for an individual object.

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

Referring to global properties

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

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

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

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

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

Changing global properties

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

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

Some global properties can be changed by other commands. For example, the **lockScreen** property can either be set directly, or changed using the **lock screen** and **unlock screen** commands. The following two statements are equivalent:

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

Saving and restoring global properties

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

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

Text Related Properties

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

Text style properties

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

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

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

textFont, textStyle, and textSize

textShift

backgroundColor and foregroundColor

backgroundPattern and foregroundPattern (Unix systems)

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

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

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

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

Field" reports "mixed".

Formatted text properties

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

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

For more information on these properties see the chapter on *Processing Text and Data*, as well as the individual entries for these properties in the *LiveCode Dictionary*.

The formattedRect and related properties

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

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

The imageSource, linkText, and visited properties

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

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

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

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

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

Creating and Deleting Objects

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

The Create Object Command

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

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

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

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

The *group* is any group that's on the current card. If you specify a *group*, the new object is a member of the group, and exists on each card that has the group. If you don't specify a group, the object is created on the current card and appears only on that card.

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

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

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

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

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

The Delete Object Command

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

```
delete {object}
```

The *object* is any available object.

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

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

Creating Objects Off-screen Using Template Objects

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

If you need to create a new object and then set some properties on the object, it is more efficient to make the changes to the template object, then create the object. Because the object can be created with all of its properties set correctly, there is no need to lock the screen and update or reposition the object after creating it. For example, the LiveCode development environment uses the template objects internally to create new objects from the main tool palette.

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

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

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

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

When you have used the `templateObject` to create a new object, you should reset it before using it again. Resetting the `templateObject` sets all off its properties back to defaults.

reset the `template[Objecttype]`

For example, to reset the `templateButton`:

```
reset the templateButton
```

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

Property Arrays using the Properties Property

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

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

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

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

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

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

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

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

Custom Properties

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

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

Using Custom Properties

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

Use a custom property when you want to:

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

Creating a Custom Property

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

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

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

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

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

The Content of a Custom Property

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

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

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

Changing a part of a property

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

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

Custom Property Names

The name of a custom property must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore.

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

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

Custom property names beginning with "**rev**" are reserved for LiveCode's own custom properties. Naming a custom property with a reserved name may produce unexpected results when working in the development environment.

Referring to Custom Properties

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

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

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

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

Nonexistent Custom Properties

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

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

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

Finding out Whether a Custom Property Exists

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

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

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

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

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

Custom Properties & Converting Text Between Platforms

When you move a stack developed on a Mac OS or OS X system to a Windows or Unix system (or vice versa), LiveCode automatically translates text in fields and scripts into the appropriate character set. However, text in custom properties is not converted between the ISO and Macintosh character sets. This is because custom properties can contain binary data as well as text, and converting them would garble the data.

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

Because of this, if your stack displays custom properties to the user--for example, if the stack puts a custom property into a field--and if the text contains special characters, it may be displayed incorrectly if you move the stack between platforms. To avoid this problem, use one of these methods:

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

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

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

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

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

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


Storing a file in a custom property

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

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

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

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

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

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

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

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

When restoring the file, decompress it first:

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

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

Deleting a custom property

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

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

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

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

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

Custom Property Sets

Custom properties can be organized into *custom property sets* – or arrays of custom properties. A custom property set is a group of custom properties that has a name you specify.

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

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

Creating custom property sets

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

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

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

The statement above creates the custom property set.

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

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

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

Custom property set names

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

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

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

Referring to custom property sets

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

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

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

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

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

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

Finding out whether a custom property set exists

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

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

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

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

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

The default custom property set

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

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

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

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

Using multiple custom property sets

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

A translation example

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

```
answer the standardErrorPrompt of this stack
```

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

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

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

```
answer the standardErrorPrompt of this stack
```

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

Copying custom properties between property sets

When it's created, a custom property set is empty, that is, there aren't any custom properties in it. You put custom properties into a new custom property set by creating the custom properties while the set is active:

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

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

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

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

Caution: Custom property sets in the development environment

Arrays, custom properties, and custom property sets

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

Referring to custom properties using array notation

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

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

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

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

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

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

Storing an array in a custom property set

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

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

```
set the customProperties["myProperty"] of me to theArray
```

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

```
get the myProperty["myElement"] of field "Example"
```

Deleting a custom property set

As described above, the **customPropertySets** property of an object is a list of the object's custom

property sets. You can set the **customPropertySets** of an object to control which custom property sets it has.

In LiveCode, there is no command to delete a custom property set. Instead, you place all the custom property set names in a variable, delete the one you don't want from that variable, and set the **customPropertySets** back to the modified contents of the variable. This removes the custom property set whose name you deleted.

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

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

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

Attaching Handlers to Custom Properties

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

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

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

- validate a custom property's value before setting it
- report a custom property's value in a format other than what it's stored as
- ensure the integrity of a collection of properties by setting them all at once
- change an object's behavior when a custom property is changed

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

Responding to changing a custom property

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

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

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

The structure of a **setProp** handler

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

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

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

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

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

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

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

Passing the setProp trigger

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

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

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

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

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

Using the message path with a setProp trigger

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

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

Note: To refer to the object whose property is being set, use the **target** function. The **target** refers to the object that first received the **setProp** trigger

- the object whose custom property is being set - even if the handler being executed is in the script of another object.

Setting properties within a setProp handler

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

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

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

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

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

Nonexistent properties

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

Custom property sets and setProp handlers

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

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

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


```

setProp myFrenchStrings[myPropertyName] newValue
-- The myPropertyName parameter contains the name of
-- the property that's being set
    switch myPropertyName
        case "standardErrorPrompt"
            set the myFrenchStrings["standardErrorPrompt"] of the target to return &
newValue & return
            exit myFrenchStrings
        break
        case "filePrompt"
            set the myFrenchStrings["filePrompt"] of the target to return& newValue &
return
            exit myFrenchStrings
        break
        default
            pass myFrenchStrings
        end switch
end myFrenchStrings

```

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

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

Suppose you change the "standardErrorPrompt" custom property:

```

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

```

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

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

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

```

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

```

is equivalent to:

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

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

Responding to a request for the value of a custom property

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

A getProp call acts very much like a custom function call. It is sent to a particular object. If that object's script contains a **getProp** handler for the property, the handler is executed, and LiveCode substitutes the value it returns for the custom property reference. Otherwise, the call travels along the message path until it finds a handler for the property. If the getProp call reaches the end of the message path without being trapped, LiveCode substitutes the custom property's value in the expression.

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

The structure of a getProp handler

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

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

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

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

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

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

Returning a value from a getProp handler

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

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

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

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

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

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

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

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

Using the message path with a `getProp` call

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

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

Nonexistent properties

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

Custom property sets and `getProp` handlers

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

When you use the value of a custom property in a custom property set, the `getProp` call is named for the set, not the property. The property name is passed in a parameter using array notation. This means that, for custom properties in a set, you write a single `getProp` handler for the set, rather than one for each individual property.

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

```

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

```

As you can see from the **pass** and **end** control structures, the name of this **getProp** handler is the same as the name of the custom property set that it controls – "expertSettings". Because there is only one handler for all the custom properties in this set, the handler uses the **switch** control structure to perform a different action for each property that it deals with.

Suppose you get the "fileMenuContents" custom property:

```

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

```

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

Virtual Properties

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

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

You can use virtual properties to:

- Give an object a set of behaviors
- Compute a value for an object
- Implement a new property that acts like a built-in property

When to use virtual properties

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

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

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

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

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

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

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

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

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

Handlers for a virtual property

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

Virtual property setProp handlers

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

pass control structure, because a virtual property should not be set. Since the property is set automatically when the trigger reaches the end of the message path, a virtual property's handler does not pass the trigger.

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

Virtual property getProp handlers

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

Creating new object properties

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

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

Setting the "percentWidth" property

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

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

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

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

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

Getting the "percentWidth" property

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

```

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

```

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

```

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

```

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

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

Limiting the "percentWidth" property

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

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

```

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

```

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

Property Profiles

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

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

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

Use property profiles when you want to:

- Create "skins" for your application

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

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

Profile names

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

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

The master profile

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

If you don't specify a property setting in a profile, the master profile's setting is used, so you don't have to specify all properties of an object when you create a profile, only the ones you want to change.

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

Switching between profiles

Switching a single object

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

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

Switching all the objects on a card

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

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

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

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

Switching all the objects in a stack

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

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


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

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

Switching all the objects in a stack file

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

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

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

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

Creating a profile in a handler

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

To enable creating profiles, check the "Create profiles automatically" box in the "Property Profiles" pane of the Preferences window. If this box is checked, setting the **revProfile** property of an object automatically creates the profile.

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

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

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

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

You can control this behavior either in Preferences window or using the **gRevProfileReadOnly** keyword. If you don't want to save property changes when switching profiles, do one of the following:

Set the **gRevProfileReadOnly** variable to true:

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

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

The two methods of changing this setting are equivalent: changing the **gRevProfileReadOnly** variable

also changes the preference setting, and vice versa.

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

Adding profile settings in a handler

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

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

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

Managing Windows, Palettes and Dialogs

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

Moving a window

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

The **location** property specifies the center of the stack's window, relative to the top left corner of the main screen. Unlike the **location** of controls, the **location** of a stack is specified in absolute coordinates. The following statement moves a stack to the center of the main screen:

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

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

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

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

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

Changing a window's layer

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

```
go stack "Alpha"
```

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

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

The palette layer

Normally, palette windows float above editable windows and modeless dialog boxes. A palette will always float above a standard window, even if you bring the standard window to the front. This helps ensure that palettes, which usually contain tools that can be used in any window, cannot disappear behind document windows. It's also a good reason to make sure you design palette windows to be small, because other windows cannot be moved in front of them if the palette blocks part of the window.

The system palette layer

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

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

The active window

In most applications, commands are applied to the active window. Since LiveCode gives you the flexibility to use several different window types, not all of which are editable, the current stack is not always the same as the active window. The current stack is the target of menu choices such as **View -> Go Next** and is the stack specified by the expression *this stack*.

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

Finding the current stack

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

1. If any stacks are opened in an editable window, the current stack is the frontmost unlocked stack. (A stack is unlocked if its **cantModify** property is set to false.)
2. If there are no unlocked stacks open, the current stack is the frontmost locked stack in an editable window.
3. If there are no stacks open in an editable window, the current stack is the frontmost stack in a modeless dialog box.
4. If there are no editable or modeless windows open, the current stack is the frontmost palette.

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

The topStack function and the defaultStack property:

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

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

Changing the current stack

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

A note about Unix systems

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

Creating a backdrop

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

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

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

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

In the LiveCode development environment, you can create a backdrop by choosing **View -> Backdrop**. Use the Preferences dialog box to specify a backdrop color to use.

Open, Closed, and Hidden Windows

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

Hidden stacks

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

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

Loaded stacks

A stack can also be loaded into memory without actually being open. A stack whose window is closed (not just hidden) is not listed by the **openStacks** function. However, it takes up memory, and its objects are accessible to other stacks. For example, if a closed stack that's loaded into memory contains a certain image, you can use the image as a button icon in another stack.

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

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

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

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

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

The states of a stack

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

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

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

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

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

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

Window Types and the Mode Property

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

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

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

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

Window Appearance

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

The metal property

On OS X systems, you use a stack's **metal** property to give the stack window a textured metal appearance. This metal appearance applies to the stack's title bar and its background.

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

Window background color

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

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

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

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

The Decorations Property

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

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

The statement above sets the stack's **minimizeBox** property to true, as well as showing its title bar, and sets other stack properties (**maximizeBox**, **closeBox**, **metal**) to false. Conversely, if you set a stack's **minimizeBox** property to true, its **decorations** property is changed to include "minimize" as one of its items. In this way, the **decorations** property of a stack interacts with its **closeBox**, **minimizeBox**, **zoomBox**, **metal**, **shadow**, and **systemWindow** properties.

The decorations property and menu bars in a window

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

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

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

Title bar

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

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

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

Window title

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

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

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

The close box

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

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

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

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

The minimize box or collapse box

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

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

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

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

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

The maximize box or zoom box

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

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

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

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

Making a stack resizable

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

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

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

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

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

Irregularly-Shaped and Translucent Windows

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

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

Programming Menus & Menu Bars

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

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

To easily create menu bars that work cross-platform, choose **Tools -> Menu Builder**. See the section on the *Menu Builder* in the chapter on *Building a User interface* for more details. The details about menu bars in this topic are needed only if you want edit menu bars by script, for example if you want to include specific features not supported by the Menu Builder.

Menu Types

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

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

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

Button Menus

You can create a button menu by dragging out one of the menu controls from the tools palette. However,

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

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

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

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

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

Handling the menuPick message

When the user chooses an item from the menu, LiveCode sends the **menuPick** message to the button. The message parameter is the name of the menu item chosen. If you want to perform an action when the user chooses a menu item, place a **menuPick** handler like this one into the button's script:

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

Changing the currently-chosen menu item

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

If you change the currently-chosen menu item in option menus, also set the button's **menuHistory** property to the line number of the newly chosen item. This ensures that the new choice will be the one under the mouse pointer the next time the user clicks the menu.

Creating Cascading Menus

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

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

First Item

Second Item**Third Item Is A Submenu****First Item In Submenu****Second Item In Submenu****Last Menu Item Not In Submenu**

The depth of a submenu item is determined by the number of tab characters before the menu item's name. The submenu item becomes part of the closest line above the submenu item that has one fewer leading tab character.

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

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

Cascading menus and the menuPick message

When the user chooses a menu item in a cascading menu, the parameter of the **menuPick** message contains the menu item name and the name of the submenu it's part of, separated by a vertical bar (|). For example, if the user chooses the "Second Item In Submenu" from the menu described above, the parameter sent with the **menuPick** message is:

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

Ticks, Dashes & Checks in Menus

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

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

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

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

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

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

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

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

Putting the (character anywhere in a line disables the menu item. To put a (character in a menu item

without disabling it, precede it with a backslash: \(.

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

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

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

Enabling and disabling menu items

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

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

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

Menu Bars on Linux and Windows Systems

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

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

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

1. Create a button for each menu, set the **style** of each button to "menu", and set the **menuMode** of the button to "pulldown". You can either set these properties in a handler, or simply choose **Object -> New Control Pulldown Menu** to create each button.
2. Put the menu items into each button's contents. In each button's script, create a **menuPick** handler to perform whatever actions you want to do when a menu item is chosen.
3. Select the buttons and form them into a group, then move the group to the appropriate position at the top of the window. For Windows systems, set the **textFont** of the group to the standard font for Windows menus, "MS Sans Serif".

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

Menu Bars on Mac OS X Systems

To create a Mac OS X menu bar, you follow the same steps as for a Linux and Windows menu bar above. This places a group of buttons, each of whose **menuMode** property is set to "pulldown", at the top of your stack window.

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

menu bar, you don't need to see them in the stack window as well.

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

The default menu bar

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

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

Button menu references

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

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

Because menus are also buttons, you can use a button reference to get the same information. But you may need to specify the group and stack the button is in, to avoid ambiguity. (For example, if there is a standard button named "Edit" on the current card, the expression `button "Edit"` refers to that button, not to the one in the menu bar.) An unambiguous button reference to a menu might look like this:

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

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

The layer of menu buttons

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

Changing menus dynamically

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

The editMenus property

When you set the **menubar** property of a stack to the name of a group, the stack is resized and scrolled up so the part of the window that holds the menus is not visible. To reverse this action so you can see,

select and edit the buttons that make up your menu bar, set the **editMenus** property to true. This resizes the stack window so the button menus are again visible, and you can use the tools in the LiveCode development environment to make changes to them.

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

Special menu items

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

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

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

When LiveCode sets up the Mac OS X menu bar, it automatically makes the last button the Help menu (regardless of the button's name). The standard Help menu items, such as the "Search" bar are included for you automatically; you don't need to include them in your Help menu button, and you can't eliminate them from the Help menu.

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

The File menu and the "Quit" menu item

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

The Edit menu and the "Preferences" menu item

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

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

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

Choosing the special menu items

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

Stack Menus

Button menus can be used for most kinds of standard menus. However, if you want to create a menu with a feature that is not supported by button menus--for example, if you want a popup menu that

provides pictures, rather than text, as the choices--you can create a menu from a stack.

Creating a stack menu

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

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

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

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

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

Displaying a stack menu

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

You can also display a stack menu without associating it with a button, by using the **pulldown**, **popup**, or **option** command. Normally, you use these commands in a **mousedown** handler, so that the menu appears under the mouse pointer:

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

Displaying Context Sensitive Menus

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

popup command: opens a stack as a popup menu

pulldown command: opens a stack as a pulldown menu

option command: opens a stack as an option menu

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

Searching and Navigating Cards using the Find Command

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

```
find [form] textToFind [in field]
```

The *form* can be one of the following:

normal

characters or character (or chars or char)

words or word

string

whole

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

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

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

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

When the find command finds a match, it highlights the match on the screen – if necessary navigating to the card that contains the match and scrolling the field so the text is in view.

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

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

```
find empty
```

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

Using Drag and Drop

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

Initiating a Drag Drop

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

If you drag from within a field, a **dragStart** message is sent. To allow drags from a locked field or from another object type, in the object's **mouseDown** handler, set the **dragData** property to the data you

want to drag. When there is a value in the **dragData**, a drag and drop is initiated when the mouse is clicked and then moved.

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

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

text	The plain text being dragged.
HTML	The styled text being dragged, in the same format as the htmlText
RTF	The styled text being dragged, in the same format as the RTFText
Unicode	The text being dragged, in the same format as the unicodeText
image	The data of an image (in PNG format)
files	The name and location of the file or files being dragged, one per line

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

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

Tracking During a Drag Drop Operation

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

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

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

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

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

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


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

Responding to a Drag and Drop

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

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

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

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

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

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

```
on dragEnd -- remove data being dragged
    delete the dragSource
end dragEnd
```

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

```
select the dropChunk
```

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

Prevent Dragging and Dropping to a Field

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

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

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

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

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

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

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

Error Handling & Debugging

Introduction

In an ideal world everyone would write perfect code and there would never be any need to debug. However in reality, virtually every project is going to require some degree of debugging. The more complex the project the more like this is to be true. Fortunately LiveCode includes a full plethora of debugging tools and techniques that make it quick and easy to track down errors. The live run-edit cycle allows you to see the effect of corrections as soon as you make them. And, unlike working in a lower-level language, when you make a mistake you will receive a human-friendly error message pointing you to where the error occurred, instead of the application unexpectedly quitting.

As well as the set of built-in error reporting and debugging tools, LiveCode also allows you complete flexibility over error handling, allowing you to provide a customized experience to the end user of your application.

Common Techniques for Solving Problems

If you encounter a problem with your code, there are a number of methods available to help you track it down. In this section we detail some of the main techniques you may find useful.

Errors During Execution

Often the first thing that will alert you to a problem is an error dialog.

Figure 80 – The Error Dialog

The *Execution Error* dialog is displayed when your script is running and encounters an error, if Script Debug Mode is turned off (under the Development menu).

In the example above, the script was attempting to access field 100. This field does not exist. When LiveCode encounters a problem like this, execution will *halt* and the dialog above will be displayed.

If you know what the error is from looking at it, use the *Script* button to go directly to the script and edit it. The line that generated the error will be highlighted in the script window.

If Script Debug Mode is on, the script editor will automatically open up on execution error and display the line which caused the error.

Errors During Compiling

A *Script Error* occurs when your script cannot be compiled because of a syntax error. This is flagged up in the script editor when you attempt to compile a change to a script by pressing the Apply button in the Code Editor. Correct the error then press the Apply button to compile the script again.

Caution: If a compile error is generated then the entire script to which it applies will not be compiled, not just the line or handler that contains the error. If other scripts are attempting to call commands or functions in this script they will not be able to run until you correct the problem and compile the script again.

Because LiveCode compiles all the scripts within a stack when it loads them, a *Script Error* dialog can be generated when you load a stack from disk for the first time – if you saved a stack that contained a script

that could not compile.

Important: Do not confuse the *Execution Error* and *Script Error* dialogs. The *Execution Error* dialog occurs when a script is running and cannot continue due to an error. The error dialog will start with the words "executing at [time]". The *Script Error* dialog appears when you attempt to compile a script that contains a syntax error. The error dialog will start with the words "executing at [time]".

Tip: If you turn on the Variable Checking option in the Code Editor, LiveCode will require that you declare all variables and enclose all literal strings in quotes. It will report a script compile error if you attempt to use an unquoted literal or do not declare a variable. This can be useful in catching mistakes before you run a script. Note that if you turn this option on and attempt to compile an existing script that has not been written in this way, it will typically generate a large number of errors that need to be corrected before you can compile it.

Suppressing Errors and Messages

If your stack gets into an unstable state where it is generating a large number of errors, you may want to temporarily turn off sending messages to the stack or displaying error messages. This can enable you to edit the stack to make the appropriate changes.

To suppress messages, press the Messages button on the toolbar or choose Suppress Messages from the Development menu. Normal system event messages will stop being sent to the stack (for example clicking a button will no longer trigger a mouseUp handler, changing card will no longer trigger an openCard handler). You can still send custom messages directly to objects within the stack.

To suppress errors, press the Errors button on the toolbar or choose Suppress Errors from the Development menu. This will prevent any errors from triggering an error display window.

Caution: Be sure to turn messages and errors back on when you have finished editing. Otherwise your stack will not work, or any error that comes up during stack operation will cause execution to halt but will not display an error message.

You can execute a single navigation command in the message box with a similar effect as Suppress Messages by including the statement `lock messages;` before it. For example, to go to the next card `lock messages; go next` or to exit LiveCode (without displaying triggering any close messages or asking to save changes) `lock messages; quit`.

Outputting information as your script executes

If you want to know about the state of a particular variable or condition during execution, you can use the *Variable Watcher*, detailed below. However sometimes you may want to run a script without opening the debugger, outputting information along the way. You can use the **put** command to output information to the Message Box, a field or a text file, the **write** command to output to the Console, or the **answer** command to display a dialog.

Output to the Message Box

The Message Box is a convenient way to output information while a script is running. It allows you to display the information you want within the IDE without having to create a field or stack to display the information. Using a put command without specifying a destination outputs to the Message Box:

```
put tVar
```

In the example above, substitute tVar with the name of the variable in your script that you want to see the contents of. You can also output information only if a certain condition is met:

```
if tVar is true then put tData
```

Whenever you display something in the message box, a special global variable called **message** (often abbreviated to **msg**) is updated. This allows you to check the contents of the message box easily, as well as add or append information rather than replacing it.

```
-- puts tInformation and a return after the data
-- already in the message box
put tInformation & return after msg
```

For more information, see the section on the *Message Box*, above.

Output to Standard Out (stdout or the Console on Mac)

Standard Output (stdout) is a useful place to log messages. Unlike the Message Box, it is easy to log a sequence of events which you can then scroll back to review later on. It also has the advantage of being external to LiveCode so using it does not interfere with your application window layering or focus in any way. On Mac, it can be accessed by opening the Console application, located in the Utilities folder within Applications.

The syntax to write something to stdout or the Console is:

```
write tMessage & return to stdout
```

Tip: If you are writing a lot of data out to the console, it can be useful to append the time to each one to make it easier to debug.

`write tMessage && the long time & return to stdout`. If you need more granularity than seconds, use the milliseconds instead of the long time.

If you are inserting debugging output statements into your code, consider making them conditional on a global variable. This allows you to turn on debugging by setting the variable without making changes to code. Even better it prevents forgotten debugging code in your application inadvertently filling the console with messages on an end user's system.

```
if gDebugging then write tMessage & return to stdout
```

Output to a field

You can create a stack that contains fields used for debugging:

```
put tVar & return after field "debugging info" of stack "my debug stack"
```

Tip: You may want to create a set of tools that makes it easy to debug a specific application you are working on. Create this stack then save it into your plug-ins folder so it is available from the Development menu in the IDE.

Output to a dialog

To display a dialog with the contents of a statement, use the `answer` command. This method is suitable if you want to output something quickly, but is usually unsuitable for larger debugging tasks as it pauses execution to display a dialog each time.

```
if tVar is true then answer tVar
```

Output to a text file

If you want to log information more permanently, you can use a text file. You may want to store the file path in a global variable so you can change it easily. See the [Transferring Information](#) guide for more information.

```
put tVar & return after URL "file:debug.txt"
```

Interrupting Execution

If you need to interrupt a script while it is running, press control-period (or command-period on Mac OS). Note that interrupting a script will only work if the global **allowInterrupts** property is set to true.

Tip: On Mac OS X systems, if your application has got into an unstable state and you are unable to interrupt it with command-period you may be able to interrupt it and regain control by sending it a signal. Open the Terminal utility and then use `top -o cpu` or `ps ax` to retrieve the process ID for LiveCode. Then run `kill -sighup [processID]` where `[processID]` is the ID of the LiveCode process.

The Debugger

Typically when you want to track down a problem with your script, you will want to use the debugger. The debugger provides a simple interface that allows you to step through your script line by line as it executes. You can see the results of each statement as it happens. If you load the Variable Watcher from within the debugger, it will show you the contents of all of the variables used in a particular handler. These values will be updated as you step through. You can even edit these values while your script is running. This can be useful if you spot a problem in a given area or want to test a section of code with a different set of data as it runs.

To activate the debugger, first ensure that Script Debug Mode is turned on in the Development menu. Then open up the script you want to debug and click in the gray bar to the left of the line where you want to open the debugger. Alternatively you can write the command **breakPoint** into the script. Using the **breakPoint** command allows you to break conditionally - `if tVar is true then breakPoint`.

Next run your script as normal. When LiveCode reaches a breakpoint it will pause execution and load up the debugger.

Important: To see the contents of variables while your script is running, wait for the debugger to open then open the *Variables* pane (or select *Variables* from the *Debug* menu).

Figure 81 – The Debugger

For a table describing each of the buttons in the debugger together with their keyboard shortcuts, see the section on The Debug Menu in the LiveCode IDE guide.

Press the Step Into button to execute the current line and move to the next line. Step Over performs the same action, except that if the current line calls a function it will run that entire function without opening it in the debugger.

Once you have found the cause of the problem, press the Run button to exit the debugger and run the script at full speed. Alternatively press the Abort button to halt execution on the current line and edit the script.

Tip: To improve the performance when debugging a complex script, set a breakpoint further down in the script during the debugging process and press the Run button instead of pressing Step Over. The debugger is temporarily deactivated until the new breakpoint is reached, which means the script runs at full speed. When using Step Over, the debugger is still active even though it does not display a routine as it runs, which is slower.

Custom Error handling

If you are creating an application that is going to be distributed, you may want to include a method for catching and handling errors. LiveCode provides two such methods. The first is the **try/catch** control structure. This control structure can be inserted around any routine that you know may encounter problems when in use. For example, you may wish to include this around a routine that reads in a file from disk to handle the case where a corrupted file has been selected. The second method is to write a custom **errorDialog** routine. Unlike a **try/catch** control structure, an **errorDialog** handler is global to your application (or card or stack within an application) and does not provide a mechanism to allow a script that encounters an error to continue. Use this method to display a custom error dialog when an error occurs that you are unable to predict and report using try/catch.

Using try/catch

Enclose code that may be error prone within a **try** control structure. The following example shows a routine that opens and reads from a file enclosed in a try statement. If an execution error occurs in any of the statements after the start of the try control structure, the catch clause will be triggered with the details of the error. In the example below we declare a variable `someError` to contain details of the error.

```
try
  open file tFile
  read from file tFile until eof
  close file
catch someError
  answer "An error occurred reading a file" && someError
end try
```

Tip: The data returned to the error routine is returned in the internal format that LiveCode uses to display execution errors. To look up the human friendly string associated with a particular error, look at the first item returned against the list of execution errors stored in the LiveCode IDE.

```
put line (item 1 of someError) of the cErrorsList of \
card 1 of stack "revErrorDisplay"
```

This will only work in the IDE.

If you want to include statements that will be run regardless of whether there has been an error or not,

include the statements as part of a **finally** clause.

To create readable error messages for cases where you anticipate there may be an error, use the **throw** keyword. For example, if we want to display an error message when the result for opening a file returns something:

```
open file tFile
if the result is not empty then throw the result
```

In the example above, if the file cannot be opened and the result is set, the value of the result will be passed to the catch statement in the `someError` variable.

Writing a custom `errorDialog` routine

When an execution error occurs, an **errorDialog** message is sent. The IDE uses this message to display the execution error dialog. However you can write and include your own custom **errorDialog** routine. This is useful if you are planning to distribute your application. For example, you could create a stack that transmits the error information directly to your technical support department or displays a custom message on screen. A basic **errorDialog** routine is shown below.

```
on errorDialog pError
answer "There was an error" && pError
end errorDialog
```

This routine will also be activated if you use the `throw` keyword to throw an error (outside of a **try/catch** control structure).

The Message Watcher

The Message Watcher lets you see what messages are sent during a particular operation. it can be useful if you want to create a log of the sequences of messages generated during a particular operation. It also logs how long each message takes to run. This makes it useful when you are looking for bottlenecks in your code. The Message Watcher is located in the Development menu,

Figure 82 – The Message Watcher Window

Message list area

Lists messages as they happen. The format is *the name* of the message, *the time* the message was sent and the number of *milliseconds* since the last message. Click to select a message, double click to open it in the Code Editor. In the example above we can see that the user was inactive for 4.7 seconds before moving the mouse over an object that triggered a *mouseenter* message. 599 milliseconds later they clicked the mouse triggering a *mouseup* message. The *mouseup* handler called *calculateResult* 0 milliseconds later, then *formatData* was called 147 milliseconds later. Because there were no user-generated messages in the sequence, we know that the *calculateResult* handler took 147 milliseconds to run.

Object field

Shows the object that the message was sent to. Click on a line in the Message List to update this field.

Message type	Shows the type for the selected message – <i>command</i> , <i>function</i> , <i>getProp</i> or <i>setProp</i>
Active	Check this box to activate the Message Watcher. Deactivate the Message Watcher when you have captured information about the sequence of events you are interested in and want to stop the logging of additional messages.
Clear	Clears the message list field.
Suppress	Allows you to set what messages you want displayed and which ones you want to suppress. Use this option to narrow down the messages that are logged so you can see information about the sequence you are interested in. See below for more information.

Figure 83 – Suppress Messages Option Screen

Action - Handled	Don't log any message that causes a handler to run when it is sent.
Action - Not Handled	Don't log any message that does not cause a handler to run when it is sent. This is the default option and prevents the log from filling up with messages that do not cause any scripts to run.
IDE Messages	Do not log LiveCode IDE messages. The LiveCode IDE generates lots of messages as it is written in LiveCode. This is the default option but you may wish to display these messages if you are customizing the LiveCode IDE.
Handler Type	Do not log the selected type of handler. For example, to prevent displaying all function calls, check the function box.
Message list	A list of messages not to log. By default <i>mouseMove</i> is listed as otherwise <i>mouseMove</i> messages may flood the display whenever you move the mouse.
Add	Add a message name to prevent it from being logged.
Delete	Delete a message name to cause it to be logged in the future.

Tracking Down Problems in Standalone Applications

We recommend that you debug your application as fully as possible in the IDE, so that you can use the Debugger and other tools. However occasionally you may need to track down a problem that only occurs in a standalone application. If this happens, use the techniques described above for writing information out to the stdOut or a text file. You can also include an error display dialog using the *Bug Reports* tab within the Standalone Settings screen. Be sure to check either the *Allow user to save report to file* or *Allow user to email report* buttons so that you can view the error generated. For more information, see the guide on *Deploying your Application*.

LiveCode Data Grid

Introduction

What is the Data Grid?

The LiveCode Data Grid enables you to integrate powerful tables and forms into your projects. Data grids combine LiveCode groups and behaviors to provide you with a simple, yet flexible means of displaying your data in just about any way you want.

A Data Grid Table

The table style of a data grid allows you to display your data in a modern looking table complete with headers, sorting, column alignment and more. By default a data grid table will use a LiveCode field to each column of each row. If you need to customize a column with graphics you can define your own templates for each column. A custom template can be any LiveCode control such as a graphic or group.

A Data Grid Form

The form style of a data grid (1) is similar to a list field (2) in LiveCode in that it allows you to display records from a data source (text file, database, XML, etc.) that the user can select.

The difference is that the data grid uses a group as a template for each record. This means that for each record you display you can create rich user interfaces using menus, images, etc.

The data grid form is also very fast as it only draws the records that are currently visible on the screen. This means you can display large records sets without slow performance.

Because the data grid form is so easy to use it is not only limited to displaying lots of records. Any list where you want a rich UI and complete control over layout and processing of engine events can benefit from being displayed using a data grid form.

Can You Show Me Some Examples?

Of course we can! The areas of the screenshots highlighted in blue are instances of the data grid control.

Data Grid Table

Data Grid Forms

How Do I Create My First Data Grid Table?

This lesson will show you how to create a bare bones data grid table. Data grid tables are useful when you need to display rows of data in structured columns.

Locate Data Grid on Tools Palette

Drag Data Grid Onto Card

Populate Data Grid Using The Property Inspector

Data Grid tables display data in columns that you can customize the look of. You don't have to worry about customization right now though because a Data Grid table can display plain text just fine. Let's test it out.

As with all controls in LiveCode you can open the Data Grid's **Property Inspector** three ways.

Select the data grid in question and then:

1. Go to the **Object** menu and select the **Object Inspector** item
2. Right (central) click on the data grid and select **Property Inspector** from the popup
3. Double click on the data grid itself.

So, open the **Property Inspector** for the data grid you have just dragged onto the card.

1. Switch to the **Contents** pane of the **Property Inspector**.
2. Type or paste some tab delimited text into the field. Click out of the field or press **ctrl-return** to save the text you entered.

The tab delimited text you entered in the field will appear as columns in the data grid (1). The **Property Inspector** automatically creates a column in the Data Grid for each column in the text you provide (2).

Customizing Columns

Now that you've populated a Data Grid Table with some data let's look at the columns in the Data Grid.

Normally when you work with a Data Grid Table you will create columns in the **Property Inspector** in advance using the "+" button (1). Seeing as the columns needed already exist you can go through and rename *Col 1* to *state* and *Col 2* to *code* (2).

Note that a label for the columns (3) has also been assigned . Labels are useful for customizing the column labels in the Data Grid Table.

Populate Data Grid Using dgText Property

Now that you have defined your columns let's look at how to populate a Data Grid Table by setting the **dgText** property. Here is an example handler (with comments) that you can place in a button.

Note: Verify that the name of your data grid matches the name used in the code below otherwise an error will occur when you try to run it.

```
on mouseUp
  ### Create tab delimited data.
  ### Note that first line has name of columns.
  ### Providing names tells Data Grid how to map
  ### data to appropriate columns.
  put "state" & tab & "code" & cr & \
  "ALABAMA" & tab & "AL" & cr & \
  "ALASKA" & tab & "AK" into theText
  ### Let Data Grid know that first line has column names
  put true into firstLineContainsColumnNames
  set the dgText [ firstLineContainsColumnNames ] of group "DataGrid" to theText
end mouseUp
```

Executing the above code would give you the following result in the Data Grid table.

Populating the Data Grid Using dgData Property

Here is an example of populating the Data Grid Table by setting the **dgData** property to an array. The end result is the same as the previous step.

Note: Verify that the name of your data grid matches the name used in the code below otherwise an error will occur when you try to run it.

```

on mouseup
  ### Create a nested array.
  ### Array key names in the 2nd dimension
  ### dictate values for individual columns
  put "ALABAMA" into theDataA[1]["state"]
  put "AL" into theDataA[1]["code"]
  put "ALASKA" into theDataA[2]["state"]
  put "AK" into theDataA[2]["code"]
  set the dgData of group "DataGrid" to theDataA
end mouseUp

```

How Do I Create My First Data Grid Form?

This lesson will show you how to create a bare bones data grid form. Data grid forms are useful when you need a less rigid layout than columns in a table provide. A data grid form gives you complete control over the look and feel of each record you display.

Locate Data Grid on Tools Palette

Drag Data Grid Onto Card

Change Style to "Form"

Select the Data Grid and open the LiveCode **Property Inspector**. Change the style to **form**.

Add Some Text

Now you can assign some text to the data grid. By default a data grid will render some basic data without you having to customize the row template. To assign some text:

1. Switch to the Contents pane of the **Property Inspector**. (1)
2. Type or paste some line delimited text into the field. (2)

Result

That's it! The data you entered in the field will appear in the data grid. Now you are ready to customize the look and feel of your records by modifying the row template.

You should note that while you can assign line delimited text to a data grid form, the data grid really works with multi-dimensional arrays under the hood. By using arrays you can store and display all different kinds of data in a data grid. Being able to assign the **Contents** property is merely a convenience.

Example: Creating a List of People

This lesson provides a low level detail, step-by-step example of creating a custom data grid form. Download the attached sample stack to see the scripts.

Attached Files

[datagrid_sampler.zip](#)

What You Will Create

This is the data grid that you will create. Each row displays a name, a title and an image.

Add Data Grid to Card

To begin, drag a data grid from the **Tools palette** onto your card.

Change Data Grid Style

Select the data grid and from the **Basic Properties** pane of the **Property Inspector** change the style of the data grid to **Form**.

The Data That Is Going to Be Displayed in Each Row

Now you are going to design the row template for the data grid form. The row template dictates how each record in the data grid will be displayed.

Before you start editing the row template you need to know about the data that will be displayed each row. This screenshot shows an excerpt from the code that populates the data grid using a multi-dimensional array (look in the card script of the example stack). Each row will have access to the data stored in the *FirstName*, *LastName*, *Title* and *ImageURL* keys of the array.

You are now going to add UI controls to the row template in order to display the data for each of these keys.

Edit the Row Template

With the data grid selected, click on the **Row Template** button in the **Property Inspector** to open the card containing row template controls.

Edit the Row Template Group

In order to customize the look you need to edit the contents of the template group.

-
1. Make sure the **Edit** tool is selected.
 2. Make sure that **Select Grouped** IS NOT active.
 3. Select the row template group.
 4. Click **Edit Group** in the LiveCode toolbar.

Rename Label Field

The default template group has a field named *Label*. Click on the left side of the gray rectangle to select it.(1)

Open the **Property Inspector** for this field and from the **Basic** pane (2) change the name of the field to *Name* (2). Switch to the **Contents** pane and assign *Name* as the content (3).

Make Name Field Bold

With the *Name* field still selected go the **Text Formatting** pane of the **Property Inspector** (1) and change the **Style** to **Bold**(2).

Add Title Field

Drag a label field from the Tools palette (1).

Name it *Title* (2) and assign *Title* to the content (3).

Change Text Alignment

Change the text alignment of the field to the left side using the **Text Formatting** pane of the **Property Inspector**.

Add an Image and Resize the Background Graphic

Now drag an image object into the group using the **Tools palette** (1). With the image selected, select the **Position** pane on the **Property Inspector** (2) and set the **width** and **height** of the image to 48 pixels (3). Also check the **Lock size and position** checkbox (4) so that any image is scaled to fit within these bounds.

Finally, select the **Background** graphic (5) and resize it so that it frames the other controls. In the example stack the **width** of the graphic is 214 and the **height** is 56.

Stop Editing the Row Template Group

You are now done adding the UI objects to the row template group. From the **Object** menu (1), select **Stop Editing Group**. You can then **Close** and **Save** the **row template**

Edit the Row Behavior

Now that you have designed the controls for each row you need to write the script that will move data into the controls and position them on the screen.

Click on the **Edit Script** button in the **Property Inspector** under **Row Behavior** to begin editing the row behavior script.

The FillInData Message

In the script that opens when you click the Row Behavior button you will find a command named **FillInData**. This is where you move data from the `pdataArray` parameter into the controls you created.

`pdataArray` is an array with the keys *FirstName*, *LastName*, *Name* and *Title*. The code that moves the values from the array into the UI controls is pretty straightforward.

1. Assign the values of the *FirstName* and *LastName* keys to the *Name* field.
2. Assign the value of the *Title* key to the *Title* field.
3. Assign the value of the *Image URL* key to the **filename** property of the image control.

You don't need to worry about positioning any of your controls in the **FillInData** command. You just need to write the code that assigns the data to the UI controls.

Copy & Paste


```

on FillInData pdataArray
--This message is sent when the Data Grid needs to populate
-- this template with the data from a record. pdataArray is an
-- an array containing the records data.
-- You do not need to resize any of your template's controls in
-- this message. All resizing should be handled in resizeControl.
-- Example:
set the text of field "Name" of me to pdataArray["LastName"] & \
    comma & space & pdataArray["FirstName"]
set the text of field "Title" of me to pdataArray["Title"]
set the filename of image "image" of me to pdataArray["Image URL"]
end FillInData

```

The LayoutControl Message

The **LayoutControl** message is where you position all of your controls. For this template you begin by positioning the image (1). Next you extend the *Name* and *Title* fields to the edge of the image (2). Finally you resize the *Background* graphic to fill the entire row rectangle.

Copy & Paste

```

on LayoutControl pControlRect
    local theFieldRect,theRect
    -- This message is sent when you should layout your template's controls.
    -- This is where you resize the 'Background' graphic, resize fields and
    -- position objects.
    -- The first thing you should do is capture 'the rect of me'.
    -- For fixed height data grid forms you can use items 1 through 4 of the rect as
    -- boundaries for laying out your controls.
    -- For variable height data grid forms you can use items 1 through 3 of the rect
as
    -- boundaries.
    -- Example:
    set the right of image "image" of me to item 3 of pControlRect - 5
    put the left of image "image" of me into theLeft
    put the rect of field "name" of me into theRect
    put theLeft - 10 into item 3 of theRect
    set the rect of field "Name" of me to theRect
    put the rect of field "title" of me into theRect
    put theLeft - 10 into item 3 of theRect
    set the rect of field "Title" of me to theRect
    set the rect of graphic "Background" of me to pControlRect
end LayoutControl

```

Set the Data of the Control

This is an example of a handler that populates the data grid with data by setting the **dgData** property.

The **dgData** property accepts a multi-dimensional array. The first dimension is an integer representing the row. The second dimension are the key/values you want to pass to each row.

You can find this handler in the card script of the example stack.

Copy & Paste

```
command uiPopulatePeople
  put "images/" into theImageFolder
  put "Lucky" into theDataA[1]["FirstName"]
  put "Day" into theDataA[1]["LastName"]
  put "Three Amigo" into theDataA[1]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[1]["Image URL"]
  put "Dusty" into theDataA[2]["FirstName"]
  put "Bottoms" into theDataA[2]["LastName"]
  put "Three Amigo" into theDataA[2]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[2]["Image URL"]
  put "Ned" into theDataA[3]["FirstName"]
  put "Nederlander" into theDataA[3]["LastName"]
  put "Three Amigo" into theDataA[3]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[3]["Image URL"]
  put "Jane" into theDataA[4]["FirstName"]
  put "Blue" into theDataA[4]["LastName"]
  put "Secret Agent" into theDataA[4]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[4]["Image URL"]
  put "Jefferson" into theDataA[5]["FirstName"]
  put "Blue" into theDataA[5]["LastName"]
  put "Secret Agent" into theDataA[5]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[5]["Image URL"]
  lock screen
  set the dgData of group "DataGrid 1" to theDataA
  ### Hilite first row
  set the dgHilitedLines of group "DataGrid 1" to 1
  unlock screen
end uiPopulatePeople
```

It is called by a button placed on the card **Populate Data Grid** with script

```
on Mouseup
  uiPopulatePeople
end Mouseup
```

The Result

After calling the **uiPopulatePeople** command the Data Grid should look something like this (1).



Once you have set the **dgData** property you can refresh the data grid to see any of the changes made to the row template. You can send the **ResetList** message to the data grid, or press the **Refresh Data Grid** button on the Basic Properties pane of the **Property Inspector** (2) to redraw the data using the

updated template.

Troubleshooting: Row Height

By default a data grid uses a fixed height for each row. If you have not set the row height the first time you populate the data grid with data then the value that is current for the data grid may not be suitable.

You can change the height of the row template by updating the **row height** in the **Property Inspector**.
(1)

Alternatively you could also simply uncheck the **Empty row height** property.(2)

Data Grid Fundamentals

What Is This "Data Grid Templates" Stack That Appeared in My Project?

A data grid relies on record templates (forms) and column templates (tables) to display data. These templates are merely LiveCode controls that will be used to visualize your data. In order to manage these controls the LiveCode IDE creates a stack named Data Grid Templates the first time you add a data grid to a card in your stack.

Data Grid Templates Stack

Open the **Project Browser** (from the **Tools** menu) and expand your stack by clicking on the "+" sign. Here you will see the newly created **Data Grid Templates** stack.

This stack (1) contains a single card (2) for each data grid you add to a stack. Each card contains the template(s) and behaviors that the data grid will use to display the data.

Data Grid Templates Card Contents

This screenshot shows some controls that might appear on a Data Grid Templates card. The **Row Template** (1) is the group that contains either all of the controls for a form or the **column templates** for a table. The IDE assigns the **Row Template** group to the **dgProps ["row template"]** property of the data grid.

The **Behavior Script** button is assigned as the behavior to the **Row Template** group and is used for data grid forms. This script controls how data is inserted into the **Row Template** controls and how those controls are positioned.

Opening the Data Grid Templates Stack

The LiveCode IDE makes it easy to locate your templates in the Data Grid Templates stack. Simply select the data grid control and click the **Row Template...** button (1) in the **Data Grid** (2) pane of the inspector.

The card containing the template(s) for the selected data grid will open.

What Is a Row Template?

The reason a data grid can be customized is because it uses "templates" to represent data. A template is merely a Livecode group that will be copied and used to draw your data on the screen. When working with data grid forms we refer to this group as a "Row Template".

Row Template

The **Row Template** is a Livecode group control. This group represents a single record in the data that you are displaying in a data grid form. This group can contain any Livecode control and the look and layout are entirely controlled by you.

How a Record Template Is Used

When the data grid form displays your data it copies the **Row Template** into itself. It makes just enough copies to cover the visible area and then inserts data into these copies of the **Row Template**. As the user uses the scrollbar to scroll through the data the same copies are used but new data is inserted. This means that the data grid form is never drawing more records than the user can actually see which means the data grid is fast.

What Is a Column Template?

Like data grid forms, data grid tables also use templates. The difference is that when working with a table you create a template to represent each column rather than a template to represent each row. We refer to these templates as **Column Templates**. A **column template** is similar to a **row template**. One difference is that a **column template** is not limited to being a group. You can use a field for a template, a graphic, a button, etc.

How a Column Template Is Used

When your data is drawn into a data grid table each **column template** is copied into the data grid as many times as necessary in order to fill the visible area. Data is then inserted into these templates as the user scrolls.

How Do I Populate a Data Grid With Data?

There are a couple of ways you can assign data to a data grid. This lesson will show you how.

Use The Property Inspector

You can use the Contents pane of the **Property Inspector** to quickly assign data to a data grid. This does not provide as many options as setting the properties mentioned below but it will get you up and running quickly.

1. Switch to the **Contents** pane of the **Property Inspector**.
2. Enter some tab delimited text into the field.

When you use the **Property Inspector** to assign data to the data grid the **Property Inspector** sets the **dgText** property of the data grid.

Set the dgText Property

The data grid works with arrays behind the scenes but in the interest of making life easier there is a **dgText** property. The syntax is as follows:

```
set the dgText [ pFirstLineContainsHeaders ] of group "Data Grid" to pText
```

pText is assumed to be a collection of data where each row is delimited by the return character and each item is delimited by a tab. You can map each item of each line in *pText* to a particular key in an array (and thus a table column) by passing in true for **pFirstLineContainsHeaders**. If true then the data grid will extract the first line of *pText* and match the values for existing internal key/column names. The default value for **pFirstLineContainsHeaders** is false.

If you set the **dgText** of a **data grid table** then all data will be imported and assigned to the appropriate column depending on the value of **pFirstLineContainsHeaders**. Normally you should set this property to **true** and provide the header that maps each item of each line to a specific column.

Note: that if **pFirstLineContainsHeaders** is true then the named columns must *already* exist in your data grid table in order to be displayed. Setting **pFirstLineContainsHeaders** to **true** does not create the headers for you.

If **pFirstLineContainsHeaders** is false then the **columns** property of the data grid is used for mapping. For example, the first item of a line of *pText* would be assigned to the column that appears on the first line in the **columns** property of the data grid. If line 1 of *pText* contains more items than there are columns in the table then new columns are added. Any new columns are named "Col 1", "Col 2", etc.

If you set the **dgText** property of a **data grid form** then the data will be imported but it is up to you to modify your **Row Template Behavior** to display the imported data correctly. If **pFirstLineContainsHeaders** is **false** then each item of each line in *pText* will be named "Label X" (where X is the item number) in the array that is passed to **FillInData**.

Set the dgData Property

The **dgData** property of a data grid is a multi-dimensional array and is the actual format that the data grid works with under the hood. The data grid expects the first dimension of the array to be integers. The number of keys in the first dimension represents the number of records being displayed in the data grid. Here is an example with five records:

```
theDataA[1]
theDataA[2]
theDataA[3]
theDataA[4]
theDataA[5]
```

You then store all of your custom data in the second dimension. The following is how you would display five records, each with a **Name** property:

```
theDataA[1][ "Name" ]
theDataA[2][ "Name" ]
theDataA[3][ "Name" ]
theDataA[4][ "Name" ]
theDataA[5][ "Name" ]
```

In the example code above there are 5 records, each with a **FirstName**, **LastName** and **Title** property. Your templates will have access to these properties when the time comes to draw the data on the screen.

Important: if you are using a data grid table then a key in the array must match the name of a column in the table in order for that variable (column) to be displayed. So if you have a column named *Image URL* in your table you must have a corresponding key named *Image URL*:

```
theDataA[1][ "Image URL" ]
```

How Do I Customize a Form's Row Template?

When you drop a data grid from the **Tools palette** onto a card a record template is created for you automatically. This template will display line delimited text but you will most likely want to customize the template. This lesson will show you how to begin customizing the template to meet your needs.

Reveal the Row Template

Select the data grid (1) and open the **Property Inspector** (2). Click on the **Row Template...** button (3) to open the card that the data grid's template is on.

The template for the data grid will open. You can now edit the controls within the **Row Template** group in order to customize the look and feel according to your needs.

Example of a Customized Template

Here is an example of a customized template. A *Name* (1) and *Title* field (2) have been added as well as a control for displaying a picture of the person (3).

Edit Behavior

After customizing the controls in your template you need to update the **behavior** associated with the template. The behavior script is where you tell the data grid what data goes in which controls as well as how to position the controls.

Click the **Edit Script...** button to open the behavior script.

Edit the Behavior Script

The default behavior script contains comments explaining what to do in each handler. The **FillInData** message is where you will move data from a record into a LiveCode control. To do this you assign keys of *pdataArray* (1) to a control (2).

Note that the first parameter passed to **FillInData** is an array representing the current row being displayed. Also note how this is different than the first parameter passed to **FillInData** for a **column template** which is the value of the column.

Important: Make sure that you always refer to controls in the template using **of me** (3). Since multiple copies of the template will be made you will have multiple controls with the same names. Using **of me** removes any ambiguity and ensures that data is displayed in the correct control.

An Example of a Customized Behavior

This is an example of a behavior that has been customized. The **dgData** of the data grid this template is associated with looks like this:

```

theDataA[1][ "FirstName" ]
theDataA[1][ "LastName" ]
theDataA[1][ "Title" ]
theDataA[2][ "FirstName" ]
theDataA[2][ "LastName" ]
theDataA[2][ "Title" ]

```

When filling in the data the *FirstName* and *LastName* values are placed into a *Name* field (1) and the *Title* value into a *Title* field (2).

This example also shows how to use **LayoutControl** to layout your **row template**. Notice how the rectangle of the **row template** is captured at the beginning (3) and then used to position all of the other elements (4).

How Do I Customize a Table's Columns?

A **column template** is nothing more than a LiveCode control that is named after a column in your table. This control is located in the **row template** group for your data grid. This lesson will discuss how to create templates for columns in a data grid table.

Use Property Inspector to Create a Column Template

In the **Property Inspector**, select the column you would like to customize (1). Click the "+" button at the bottom of the pane to create a **column template** (2).

Edit the Row Template Group

In order to customize the look you need to edit the contents of the template group. Select it (1) and click **Edit Group** (2) in the LiveCode toolbar.

Edit Column Group

Now select the column group (1) and click **Edit Group** again (2). At this point you can customize what controls appear in the column template.

A Column Template

In this example there are three controls for three different columns in the data grid table.

A field named *Time* that formats data into a time format (1) (Just rename the field that is included in the row template by default). A group named *My Rating*. This group has some widgets that display images of a star. (2) * A *button* named *Genre** that offers different Genre selections. (3)

Here is what those controls look like in the **Project Browser**. Notice how all of the controls are located in the **Row Template** group.

Supporting Controls

The controls created (*Time* field, *Genre* button and *My Rating* group) each have behaviors associated with them. These were created automatically and are stored in appropriately named buttons.

Behavior Example

Here is what the **FillInData** handler looks like for the *My Rating* column (as stored in the **My Rating Behaviour** button). *pData* (1) contains the value of the *My Rating* column for the row being displayed.

Note how this parameter differs than the parameter sent to **FillInData** for a **row template**. For a **row template** the parameter passed in is an array.

You then use that value to determine how many stars to display (2).

Result

Here is what the table looks like now that the custom column templates have been defined. The table found controls named *Genre*, *Time* and *My Rating* in the record template group so those were used to render the data for those three columns.

Working With Data Grids (Forms & Tables)

How Do I Determine the Selected Line?

The dgHilitedLines Property

You can get the selected line or lines of a data grid using the **dgHilitedLines** property.

Determining the Line in a Row's Behavior

If we need to determine the line number of a row's behavior script we can access the **dgLine** property of the row template. The data grid automatically assigns the **dgLine** and the **dgIndex** property when displaying the row.

Note: that the use of *of me* (1) can be used because the **mouseDown** handler is in the **row template** group behavior script (which is as if the script were in the **row template** group itself). If you were to put the **mouseDown** handler in the script of a button that was located in the row template group then you would have to use *of the dgControl of me* instead.

How Do I Get Data Associated With a Row or Column?

This lesson will show you how to get the array of data associated with a row in a Data Grid form as well as how to get the data associated with a row's column in a Data Grid table.

To get the array of data from a Data Grid you can use one of the following properties:

- the **dgDataOfIndex**
- the **dgDataOfLine**

Both properties return the array associated with a row in the Data Grid, they just allow you to target the row differently. **dgDataOfLine** allows you to target the row in the order it appears visually in the Data Grid. **dgDataOfIndex** allows you to target a row based on the internal index number that the Data Grid uses to identify the row. This value does not change, even if you change the sort order for the Data Grid.

If you just need a particular column from a row then you can use one of the following functions:

- **GetDataOfIndex()**
- **GetDataOfLine()**

Both of these functions return the value associated with a particular key, or column, for a row.

Let's look at some examples of how to use these properties and functions.

Getting Data Associated With the Selected Row

If you want to get the data associated with the selected row use the **dgHilitedLines** property. The value of that property can be used in conjunction with the **dgDataOfLine** property. Here is an example of a script that could be placed in a button. The Data Grid group script is NOT in the message path for this example:

```

on mouseUp pBtnNum
  if pBtnNum is 1 then
    put the dgHilitedLines of group "DataGrid 1" into theLine
    put the dgDataOfLine[theLine] of group "DataGrid 1" into theDataA
    ## theDataA is now an array variable.
    ## In the case of the Data Grid pictured above the keys of the
    ## array are id, FirstName, LastName and Title.
    ## This answer dialog will display: Dusty Bottoms
    answer theDataA["FirstName"] && theDataA["LastName"]
  end if
end mouseUp

```

Now let's assume that you only want to get the **id** value for the selected row without fetching the entire array. You can use **GetDataOfLine** to accomplish this. You need to keep in mind that **GetDataOfLine** (and **GetDataOfIndex**) is a function defined for a Data Grid. That means that the Data Grid group **MUST** be in the message path. If the Data Grid group is not in the message path you would need to use the **call** command.

To illustrate how to use **GetDataOfLine** let's look at how you could define your own custom property for a Data Grid. The following code defines a **uSelectedID** custom property that returns the **id** of the selected row. This code would be placed in the Data Grid group script.

```

getProp uSelectedID
  put the dgHilitedLines of me into theLine
  return GetDataOfLine(theLine, "id")
end uSelectedID

```

You can now access this custom property from any script in your application:

```

put the uSelectedID of group "DataGrid" into theSelectedID

```

Getting Data When the Selection Changes

When the user makes a new selection in a Data Grid the **selectionChanged** message is sent to the Data Grid. The following **selectionChanged** message could appear in the script of your Data Grid group:

```

-- example that would be placed in the data grid script.
on selectionChanged pHilitedIndex, pPrevHilitedIndex
  put the dgDataOfIndex [ pHilitedIndex ] of me into theDataA
  ## Now call handler in card script that loads info for selected person.
  uiViewRecordOfID theDataA["id"]
end selectionChanged

```

Getting Data in a Row Behavior

If you want to get the data in a **row behavior** script you can use the **dgIndex** or **dgLine** custom properties of the row control. Here is an example that could be used in a Data Grid's **form** behavior script:

##Data Grid Row behavior script

```
on mouseUp pMouseButtonNum
  if pMouseButtonNum is 1 then
    ## the dgIndex is a custom property of this row.
    ## the dgControl is a custom property of the data grid itself.
    put the dgDataOfIndex[ the dgIndex of me ] of the dgControl of me into theDataA
    uiViewRecordOfID theDataA["id"]
  end if
end mouseUp
```

If you wanted to move this same **mouseUp** code into the Data Grid script itself then you would change the **me** references to **target** and add a check to ensure that the user clicked on a row. This example also uses **GetDataOfIndex** rather than **dgDataOfIndex** to show an alternative.

##Data Grid Form Example: Data Grid Script

```
on mouseUp pMouseButtonNum
  if pMouseButtonNum is 1 then
    ## the dgIndex is a custom property of the row.
    put the dgIndex of the target into theIndexThatWasClickedOn
    if theIndexThatWasClickedOn is not empty then
      put GetDataOfIndex(theIndexThatWasClickedOn, "id") into theID
      uiViewRecordOfID theID
    end if
  end if
end mouseUp
```

Getting Data in a Column Behavior

If you want to get the data in a column behavior script you can use the **dgIndex** or **dgLine** and **dgColumn** custom properties of the column control. Here is an example that could be used in a Data Grid column behavior script:

Data Grid Table Example: Column Behavior

```

on mouseUp pMouseButtonNum
  if pMouseButtonNum is 1 then
    ## the dgIndex is a custom property of the row.
    ## the dgColumn is a custom property of the column.
    put GetDataOfIndex(the dgIndex of me, the dgColumn of me) into theColumnValue
    ## Do something with column value..
  end if
end mouseUp

```

If you wanted to move this same **mouseUp** code into the Data Grid script itself then you would change the **me** references to **target** and add a check to ensure that the user clicked on a column:

Data Grid Table Example: Data Grid Script

```

on mouseUp pMouseButtonNum
  if pMouseButtonNum is 1 then
    ## the dgIndex is a custom property of the row.
    ## the dgColumn is a custom property of the column.
    put the dgColumn of the target into theColumn
    if theColumn is not empty then ## User clicked on a column
      put GetDataOfIndex(the dgIndex of the target, theColumn) into theColumnValue
      ## Do something with column value..
    end if
  end if
end mouseUp

```

How Do I Add a Row of Data to an Existing Data Grid?

Sometimes you may want to add a row of data to a Data Grid without having to set the **dgData** or **dgText** property. You can add a single row of data by calling the **AddData** or **AddLine** commands of a Data Grid.

Using AddData

To use **AddData** you create an array containing the values for the new row. Here is an example of how to add a new row and have it appear as line 1 in a Data Grid.

Note: This example script resides outside of the Data Grid group script so **AddData** is not in the message path. This is why the dispatch command is used.

```

put "First Name" into theDataA["FirstName"]
put "Last Name" into theDataA["LastName"]
put "Title" into theDataA["Title"]
put 1 into theLineNo
dispatch "AddData" to group "DataGrid" with theDataA, theLineNo
put the result into theNewIndex -- integer if successful, error string otherwise

```

Using AddLine

To use **AddLine** you create a tab delimited string of text containing the values for the new row. You also need to tell the Data Grid the names of the columns that the data should map to. Here is an example of how to add a new row and have it appear as the last line in a Data Grid.

```
put "First Name" & tab & "Last Name" & tab & "Title" into theRowData
put "FirstName" & cr & "LastName" & cr & "Title" into theDataColumns
put the dgNumberOfLines of me + 1 into theLineNo
AddLine theRowData, theDataColumns, theLineNo
```

Note: This example script resides inside the Data Grid group script so **AddLine** is in the message path.

Scrolling Data Into View and Getting the Data Control

After you add the data to the Data Grid you may want to scroll the new row into view. You can call **ScrollIndexIntoView** or **ScrollLineIntoView** to do this.

```
ScrollIndexIntoView theNewIndex
```

or

```
ScrollLineIntoView theLineNo
```

How Do I Update Data in a Row?

This lesson will show you how to update data in a row.

Updating a Row's Data and Refresh Data Grid Automatically

You can update data in a row by setting the **dgDataOfIndex** or **dgDataOfLine** properties. You can set either property to an array containing the values for the row.

```
put "New First Name" into theDataA["FirstName"]
put "New Last Name" into theDataA["LastName"]
put "New Title" into theDataA["Title"]
set the dgDataOfIndex[ the dgHilitedIndex of group "DataGrid" ] \
    of group "DataGrid" to theDataA
```

The data grid will now refresh with the new data you assigned to the highlighted index.

Updating a Row's Data Without Refreshing the Data Grid

If you want to update the data in a row without automatically refreshing the data grid then you can use **SetDataOfIndex**. You can then use the **RefreshList** command to redraw the data grid.

Example that sets all values at once:

```
put "New First Name" into theDataA["FirstName"]
put "New Last Name" into theDataA["LastName"]
put "New Title" into theDataA["Title"]
dispatch "SetDataOfIndex" to group "DataGrid" \
with the dgHilitedIndex of group "DataGrid", empty, theDataA
dispatch "RefreshIndex" to group "DataGrid" with the \
dgHilitedIndex of group "DataGrid"
```

Example that sets individual keys of the row one at a time:

```
dispatch "SetDataOfIndex" to group "DataGrid" with the \
dgHilitedIndex of group "DataGrid", "FirstName", "New First Name"
dispatch "SetDataOfIndex" to group "DataGrid" with the \
dgHilitedIndex of group "DataGrid", "LastName", "New Last Name"
dispatch "SetDataOfIndex" to group "DataGrid" with the \
dgHilitedIndex of group "DataGrid", "Title", "New Title"
dispatch "RefreshIndex" to group "DataGrid" with the \
dgHilitedIndex of group "DataGrid"
```

How Do I Clear Data From a Data Grid?

Clearing the data out of a Data Grid is as easy as setting the **dgData** (or **dgText**) of the Data Grid to **empty**.

```
set the dgData of group "DataGrid" to empty
```

How Do I Add a mouseDown Event to the Data Grid Without Breaking It?

This lesson will show you how you to write your own **mouseDown** event in a Data Grid without breaking the default Data Grid behavior. You need to know how to do this when showing a contextual menu or if during the **mouseDown** event you want to use the data in the row the user clicked on.

When you add a **mouseDown** handler to a Data Grid you are intercepting a message that the Data Grid normally handles. Doing so changes the behavior of the Data Grid and you need to take that into account when coding your **mouseDown** handler.

What Doesn't Work

If you were to place the following code in your Data Grid script you would not get the result you were expecting. The reason is that the line that was clicked on would not be selected until after your popup menu was displayed. Why? Because the Data Grid behavior script processes **mouseDown** AFTER the **mouseDown** handler you have defined in the Data Grid script itself.

```

on mouseDown pMouseButtonNum
  if pMouseButtonNum is 3 then
    ## Oops! Line that user clicked on has not been selected since
    ## Data Grid has not processed mouseDown yet. Contextual
    ## menu won't target proper line.
    popup button "MyContextualMenu"
  end if
  pass mouseDown
end mouseDown

```

What Does Work: dgMouseDown

In order to work around this the Data Grid wraps all **mouseDown** functionality in a handler named **dgMouseDown**. You can call this handler in your code in order to get the expected results.

The following examples show how you can define a **mouseDown** handler in a Data Grid script. In the examples the line the user clicks on will be selected before the custom **mouseDown** code executes. Note that the **mouseDown** message is not passed after calling **dgMouseDown**. This would only repeat the call to **dgMouseDown** which is not the desired behavior.

Example: Contextual Menu

```

on mouseDown pMouseButtonNum
  ## Let Data Grid process mouseDown and select row that was clicked on
  dgMouseDown pMouseButtonNum
  ## Now contextual menu will act on the proper line.
  if pMouseButtonNum is 3 then
    popup button "MyContextualMenu"
  end if
  ## Don't pass mouseDown
end mouseDown

```

Example: Getting Value From Cell Clicked On (Data Grid table)

```

on mouseDown pMouseButtonNum
  ## Let Data Grid process mouseDown and select row that was clicked on
  dgMouseDown pMouseButtonNum
  ## Get value of column clicked on. The column name can be accessed in the
  ## dgColumn custom property of the column control (the target).
  put GetDataOfIndex(the dgHilitedIndexes of me, the dgColumn of the target) into
  \
    theColumnValue
  ## Don't pass mouseDown
end mouseDown

```

How Can I Store an Option Menu Value When the User Makes a Selection?

This lesson will demonstrate how to update the data associated with a row in a data grid when the user makes a selection from an option menu.

Example Data Grid

Here is the data grid we will be working with. *Col 2* has been given the name *Rating* and has been customized with an **option menu**. What we are going to do is update the data associated with a row to reflect the selection the user makes in the **option menu**.

Edit Column Behavior

We need to customize the column behavior. From the **Columns** pane (1) in the **Property Inspector** select the *Rating* column (2) and then click the **Column Behavior** button (3).

FillInData

In the **FillInData** handler we are setting the **menuhistory** based on the value of *pData* that is passed in. As there is only a single button in the *Rating* column template, it can be referred to as **button 1 of me**.

menuPick

A **menuPick** handler needs to be added to the column behavior script in order to accomplish the goal. When the user makes a selection from the option menu the **menuPick** message is sent by the engine. The **SetDataOfIndex** command can then update the value for the column. The parameters for **SetDataOfIndex** are the **index**, the **column name** and the **new value**. Since the above script is in a column behavior we can use the **dgIndex of me** (1) for the **index** and the **dgColumn of me** (2) for the **column name**.

The Behavior in Action

Here is what the data grid's internal array looks like before making a menu selection.

After making a selection for row 2 (1) the internal value was updated (2).

How Do I Refresh a Data Grid After Making Changes to a Template Through Script?

The command **ResetList** redraws a data grid after having copied in fresh copies of any templates. Here is an example of how you might use it.

```
## Get reference to group that serves as row template for data grid
put the dgProps["row template"] of group "DataGrid" into theRowTemplate

## For tables get reference to custom template for column
put the long id of group "My Column" of theRowTemplate into theColTemplate

## Make any updates to template
set the text of button 1 of theColTemplate to "New Option1" & cr & "New Option 2"

## Refresh the data grid
dispatch "ResetList" to group "DataGrid"
```

Or you can simply issue it from the message box.

How Do I Use a Template in Multiple Data Grids?

If your application needs to use the same style of data grid in multiple places in your project then you can easily share the same row template between multiple data grids. This lesson will show you how.

Create First Data Grid

The first step is to create your first data grid and customize the template (1). The data grid property that determines the **Row Template** that is used to draw the data grid is **dgProps["row template"]**. The message box shows a query for this property in order to get a reference to the data grids row template (2).

Create Other Data Grids

Now you can add other data grids to your project. Here the second data grid is added to the same stack as the first. After adding the data grid you can set the **dgProps["row template"]** property to the **long id** of the first data grid's **row template**.

You can also see that the **Row Behavior** for *datagrid 2* is listed as coming from the same template card as for the original *DataGrid 1* which was obtained in the previous step.

Alternatively you could also do the following: (here we are setting the properties of a data grid in a second stack)

```
set the dgProps["row template"] of group "DataGrid 1" of stack "Untitled 1" \
to the dgProps["row template"] of group "DataGrid 1" of stack "my program"
```

Note: The LiveCode IDE creates a card in the **Data Grid Template XXX** stack with a **row template** every time you drag a data grid on to a card. You may want to delete this card if you aren't going to be using that template.

Result

See how the **Row Behavior** script (1) correctly identifies the template from the first stack (**Data Grid Sampler**) and how this indeed does contain the **Row Template** (2) being used in the second stack.

How Can I See What the Data Grid's Internal Array Currently Looks Like?

Using PrintKeys

A data grid has a helper command that will print off the first line of each key in the internal data array. If you need to quickly see what the internal array looks like just send **PrintKeys** to the data grid. This will print the array in the message box.

Note: You should not rely on **PrintKeys** for anything other than taking a quick peek at the array. It only prints the first line of each key so it is not a 100% accurate view of the data.

How Do I Get Aggregate Values for Columns?

This lesson will show you how to add a custom property to a data grid that returns the aggregate value of a column. Usually this would be used with data grid tables but works equally as well with data grid forms.

Add a getProp to Your Data Grid Script

The simplest way to calculate aggregate values is to add a **getProp** handler to your data grid script. In this example a custom property called **uSumOfColumn** (1) has been defined. Passing it a column name (2) returns the sum of all rows of that column (3).

Using the Custom Property

Here is an example of using the custom property. When clicking on the button (1) the text of another field (in this case "sum") is set to the **uSumOfColumn** custom property (2).

The Result

Here the result has been put into the *Sum* field.

How Do I Determine if the Data Grid Has Focus?

You can determine if the data grid has focus by performing the following check:

```
if the long id of group "DataGrid" is in the long id of the focusedobject then
## Data grid has focus
end if
```

How Do I Export Data From a Data Grid?

This lesson will show you how to get data out of a data grid.

The Example Data Grid

The data will be exported from a data grid that looks like this.

The Handler for Populating Data Grid

```

command uiPopulatePeople
  put "images/" into theImageFolder

  put "Lucky" into theDataA[1]["FirstName"]
  put "Day" into theDataA[1]["LastName"]
  put "Three Amigo" into theDataA[1]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[1]["Image URL"]

  put "Dusty" into theDataA[2]["FirstName"]
  put "Bottoms" into theDataA[2]["LastName"]
  put "Three Amigo" into theDataA[2]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[2]["Image URL"]

  put "Ned" into theDataA[3]["FirstName"]
  put "Nederlander" into theDataA[3]["LastName"]
  put "Three Amigo" into theDataA[3]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[3]["Image URL"]

  put "Jane" into theDataA[4]["FirstName"]
  put "Blue" into theDataA[4]["LastName"]
  put "Secret Agent" into theDataA[4]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[4]["Image URL"]

  put "Jefferson" into theDataA[5]["FirstName"]
  put "Blue" into theDataA[5]["LastName"]
  put "Secret Agent" into theDataA[5]["Title"]
  put theImageFolder & "monkey.jpg" into theDataA[5]["Image URL"]

  lock screen
  set the dgData of group "DataGrid 1" to theData

  ## Hilite first row
  set the dgHilitedLines of group "DataGrid 1" to 1
  unlock screen
end uiPopulatePeople

```

This is the code that was used to populate the data grid. This shows you the keys that each record has (*FirstName*, *LastName*, *Title* and *Image URL*).

Export Handler

```

command uiExportData
  ## Export data to XML

  ## Get Data Grid Array
  put the dgData of group "DataGrid 1" into theDataA

  ## Get indexes in proper order
  put the dgIndexes of group "DataGrid 1" into theIndexes

  ## Prefix XML
  put "<people>" & cr into theXML

  ## Loop through data, putting into XML format
  repeat for each item theIndex in theIndexes
    put "<person>" & cr after theXML
    put "<first_name>" & theDataA[theIndex]["FirstName"] & "</first_name>" & cr
  after theXML
    put "<last_name>" & theDataA[theIndex]["LastName"] & "</last_name>" & cr
  after theXML
    put "<title>" & theDataA[theIndex]["Title"] & "</title>" & cr after theXML
    put "</person>" & cr after theXML
  end repeat

  ## Close root XML tag
  put "</people>" after theXML

  put theXML
end uiExportData

```

This is an example of how to get data out of a data grid. Begin by getting the **dgData** array and the **dgIndexes**. The indexes are a comma delimited list of the keys of the **dgData** in the proper order.

After you have the array and the ordered list of indexes you can loop through each record in the array. In this example the data is just wrapped in XML tags.

Example Output

This is what the output for this example looks like in the message box.

How Do I Work With Checkboxes in a Data Grid?

This lesson will demonstrate how to associate the hilite state of a checkbox in a data grid with a row value in the data grid.

Attached Files

[Checkbox_Example.livecode](#)

Adding a Checkbox to a Form

Before you begin, drag a data grid onto a stack and set the style to **Form** using the **Property Inspector**.

Click the **Row Template** button to open the template for the data grid.

Edit Row Template Group

Now that the card with the data grid row template is visible, open the **Project Browser** selecting the **Tools > Project Browser** menu option.

Select the card with your template (it should be the second card under the Data Grid Templates stack) (1). Next, double-click the **Row Template** group on the card (2). Doing so will select the group on the card (3).

Now select the **Object > Edit Group** menu option. This will put the group in edit mode which will allow you to drag controls from the tools palette into the group.

Add a Checkbox

From the **tools palette**, drag a **checkbox** onto the card and place it in the upper-left corner. The button will appear in the list of controls in the **Project Browser**.

Edit Checkbox Properties

Open the button's Property Inspector, select the **Position** pane and change the **height** to 21. Set the **left** and **top** properties to 0.

Delete the Label Field

Now that you have added the checkbox button you no longer need the *Label* field in the row template. We will do this using the **Project Browser**

Note: You need to stop editing the group before returning to the **Project Browser** in order for it to correctly update. Click on the card area under the **checkbox** and then click the **Stop Editing** button on the **Toolbar**.

You may also need to **collapse** the card **tree** in the **Project Browser** window and reopen it to force the update to the **Project Browser** display.

In the **Project Browser** select the *Label* field and delete it by pressing the **delete** key on the keyboard.

You will be asked to confirm you wish to delete this object. Check that it is identifying the correct object to delete, and then click **OK**.

After deleting the field you should have only the *Background* graphic and the *Check* checkbox in the **Row Template** group.

Edit Row Behavior

Next you need to edit the **Row Behavior** in order to take into account the new checkbox control.

Click the **Edit Script** button.

Edit the FillInData Handler

In the **FillInData** Handler you need to remove references to the *Label* field and add code for the *Check* button. The code will set the **label** of the checkbox to the **label** property of the data grid row. It will set the **hilited** property to the **checked** property of the row.

Copy & Paste

```
on FillInData pdataArray
  -- This message is sent when the Data Grid needs to populate
  -- this template with the data from a record. pdataArray is an
  -- an array containing the records data.
  -- You do not need to resize any of your template's controls in
  -- this message. All resizing should be handled in resizeControl.
  set the label of button "Check" of me to pdataArray["label"]
  set the hilited of button "Check" of me to pdataArray["checked"]
end FillInData
```

Update LayoutControl and ResetData

The **LayoutControl** and **ResetData** handlers also have references to the *Label* field which no longer exists. Update both of those handlers with references to the *Check* button.

Copy & Paste


```

on LayoutControl pControlRect
    local theFieldRect
    -- This message is sent when you should layout your
    -- template's controls.
    -- This is where you resize the 'Background' graphic,
    -- resize fields and position objects.
    -- For fixed height data grid forms you can use items 1
    -- through 4 of pControlRect as boundaries for laying
    -- out your controls.
    -- For variable height data grid forms you can use
    -- items 1 through 3 of pControlRect as boundaries,
    -- expanding the height of your control as needed.
    -- Example:
    put the rect of button "Check" of me into theFieldRect
    put item 3 of pControlRect - 5 into item 3 of theFieldRect
    set the rect of button "Check" of me to theFieldRect
    set the rect of graphic "Background" of me to pControlRect
end LayoutControl

on ResetData
    -- Sent when data is being emptied because the control is
    -- no longer being used to display data
    set the text of button "Check" of me to empty
    set the hilite of button "Check" of me to false
end ResetData

```

Update Data Grid Row Value When User Changes Checkbox State

When the user clicks on the checkbox the hilited state will change. You need to update the row data in the data grid when this happens so that the **dgData** array properly represents what is being seen in the interface. You can do this in the **mouseUp** handler of the behavior script.

During **mouseUp** the hilited state of the checkbox has been changed and so it is safe to save the value. You just need to call **SetDataOfLine** and pass in **the hilite of the target** (the target being the checkbox button).

Add the following to the behavior script. Make sure and compile the script afterwards.

Copy & Paste

```

on mouseUp pMouseBtnNum
  if pMouseBtnNum is 1 then
    ## did they click on the checkbox?
    if the short name of the target is "Check" then
      ## Update internal value in data grid
      SetDataOfLine the dgLine of me, "checked", the hilite \
        of the target
    end if
  end if
end mouseUp

```

Populate the Data Grid

You have now finished configuring the Data Grid so it is time to test. Add a button to the stack and name it **Populate grid**.

With the **Edit tool** active, *right-click* on the button and select **Edit Script**.

Add a simple script that will populate the data grid with two rows. Each has the **label** and **checked** properties that were referenced in the behavior script earlier.

Paste this script to the button script.

Copy & Paste

```

on mouseUp
  put "Line 1" into theDataA[1]["label"]
  put "true" into theDataA[1]["checked"]
  put "Line 2" into theDataA[2]["label"]
  put "false" into theDataA[2]["checked"]
  set the dgData of group "DataGrid 1" to theDataA
end mouseUp

```

Test

After compiling the script, click on the **Populate Grid** button. The card should now look similar to this.

Try it out. If you click on a checkbox in line 1 it should uncheck. Click on it again to check it.

Checking/Unchecking All Checkboxes in a Data Grid

When working with lists that have checkboxes it is nice to include the option to check or uncheck all of the items in the list. Let's look at how to do that.

Add a button to the card and name it **Check/Uncheck**.

Edit the script of the button.

The code for checking/unchecking all items in the list isn't too complicated. Here is how the logic is broken up:

1. Use the *checked* value of row 1 to decide what to set all of the other rows to. Get the current value of row 1 and toggle it.
2. Loop through all lines of the data grid, setting the value of the **checked** property of each row to the new value. Use **SetDataOfLine** as this command will not redraw the data grid each time it is called.
3. Finally refresh the data grid so that the new row values are used to display each row.

Copy & Paste

```
on mouseUp
## Get Checked Value for 1st Row ##
dispatch function "GetDataOfLine" to group "DataGrid 1" with \
1,"checked"
put the result into theCheckedValue

## Get inverse value of 1st row
put not theCheckedValue into theCheckedValue

## Update all rows
## SetDataOfLine does not redraw data grid
repeat with theLineNo = 1 to the dgNumberOfLines of \
group "DataGrid 1"
dispatch "SetDataOfLine" to group "DataGrid 1" with \
theLineNo, "checked", theCheckedValue
end repeat

## Update datagrid display
dispatch "RefreshList" to group "DataGrid 1"
end mouseUp
```

Test

Click the **Check/Uncheck** button to test. You should see the checkboxes toggle between the checked and unchecked state.

If you get the **dgData** or **dgText** property of the data grid then you will see the proper values for the *checked* state of each row.

Working With Data Grid Tables

How Do I Change Column Alignment?

Using the Property Inspector

Select the data grid and open the **Property Inspector**. Navigate to the **Columns** pane (1), select the column you want to modify (2), and change the **alignment** (3).

Using Script

You can change the **alignment** by setting the **dgColumnAlignment** for a column.

```
set the dgColumnAlignment["Name"] of group "DataGrid" to "left"
```

How Do I Sort by a Column?

Using the Property Inspector

Select the data grid and open the **Property Inspector**. Navigate to the **Columns** pane (1), select the column you want to modify (2), and check the **Sort by column** checkbox (3). You can then adjust the sort options as you see fit.

Using Script

You can sort by column setting the **dgProp["sort by column"]** property of the data grid to the name of the column you want to sort by.

```
set the dgProps["sort by column"] of group "DataGrid" to "Name"
```

Similarly you can change other properties related to the sort by script too.

How Do I Resize Columns?

Using the Property Inspector

Select the data grid and open the **Property Inspector**. Navigate to the **Columns** pane (1), select the column you want to modify (2), and set the **width** of the column using the **Width** text entry field (3).

Using Script

You can set the size of a column by setting the **dgColumnWidth** property for the column to an integer.

```
set the dgColumnWidth["Name"] of group "DataGrid" to 150
```

How Do I Override the Default Behavior for Rendering Data to a Cell?

By default, a data grid table uses a single field object for each cell in a table and assigns the **text** property of that field to the cell's data. This lesson will show you how to quickly create your own script that determines how data is rendered in the default table cell. This can be useful for rendering HTML and unicode text, trailing off text that is too wide for a column or for coloring particular cells.

Begin With a Data Grid Table

Create a Button

The **default column behavior** property can be set to a button. The script of the button will be used to fill in each cell in the table.

Drag a button onto the card and name it **My Default Column Behavior**.

Set the Script of the Button

When creating your own **default column behavior** it is a good idea to start with the script that the

data grid itself uses as the default. You can copy the data grid default script easily enough by selecting the button (1) and executing the following statement in the **Message Box** (2):

Careful of continuation character "\"

```
set the script of selobj() to the script of button "Default Column" \
  of stack "revDataGridLibrary"
```

Customize Behavior

Now this behavior can be customized. Here is what the default behavior looks like.

Important: If you decide to write a default behaviour script from scratch make sure to include the **dgDataControl** getProp handler. This is required in order for the data grid to work properly.

```
getprop dgDataControl
-- Required by library so that it can locate the control.
return the long ID of me
end dgDataControl
```

Set 'Default Column Behavior' Property

Now you can set the **default column behavior** property of the data grid. Select the button (1) and execute the following in the **Message Box** (2):

Note: Careful of continuation character "\"

```
set the dgProps["default column behavior"] of group "DataGrid 1" \
  to the long id of selobj()
```

Now it is time to customize your script!

Example: Truncate Tail

The following example will show you how to truncate the tail end of every cell whose content is too wide to fit. The data grid provides a helper command named **TruncateTail** that takes the **short id** of a field and a string that signifies the text is being truncated. A call to **TruncateTail** has been added to the **FillInData** and **LayoutControl** handlers so that cell contents are truncated when drawn or when a column is resized.

Note: **TruncateTail** works fairly well for most cases but can cause visual lag if there are lots

of cells being displayed that use **TruncateTail**. You should test your data and table to make sure it performs adequately for your needs.

Quick Tip: You can determine the name of the column that is being rendered by checking the **dgColumn** property. You could use this property if you only wanted to truncate the text in certain columns:

```
switch the dgColumn of me
  case "Col 1"
  case "Col 2"
    TruncateTail the short id of me, "...
  break
end switch
```

Refresh

To see the results, click the **Refresh Data Grid** button in the Property Inspector (1). Notice how cell contents are no truncated as needed (2).



Example: Coloring Cells



This example will dim any cell that is empty. Since the default cell consists of a single field object this can be accomplished by setting the field's **opaque** to **true**, setting the **backgroundcolor** and changing the **blendlevel**.

Result

Deleting some entries in the data and click on the **Refresh Data Grid** button to see the result



How Do I Determine if a User Clicks in the Table Header?

This lesson will show you how to determine if the user clicked in the header of a Data Grid table by showing you how to use the **dgHeaderControl** and **dgHeader** properties of the target control that was clicked on.

The Table Header



A Data Grid table has a header where the column names are displayed (1). When a user clicks in the header with the mouse the **dgHeader** of the target returns the **long id** of the group control that contains all of the header controls.

When the user clicks on a column header (2) the **dgHeaderControl** of the target returns the **long id** of

the group control that contains all of the column header controls.

Example

Here is some sample code that prints off the **dgHeaderControl** and **dgHeader** properties when the user clicks on a Data Grid. Place this script in the Data Grid script.

Copy & Paste

```
on mouseDown pBtnNum
  put "Column header control:" && the dgHeaderControl of the target \
    & cr & cr & "Header control:" && the dgHeader of the target
end mouseDown
```

After adding the script to the Data Grid the value of the **dgHeaderControl** and **dgHeader** properties are displayed in the **Message box**. In this example both properties return the **long id** of a control (1) because the click was on a column header (2) (*State* in this case.)

In the following example only the **dgHeader** property returns the **long id** of a control (1) because even though the click was in the table header it was not on a column header (2).

How Do I Display a Contextual Menu When the User Clicks on a Column Header?

Overview

This lesson will show you how to display a contextual menu (1) when the user clicks on a column header in a Data Grid table (2).

The Code

Here is some LiveScript code that will display a contextual menu. The code is located in the Data Grid's script.

A contextual menu is displayed when during the **mouseDown** message when the user clicks with the right mouse button (1).

To determine if the user clicked on a column header you can check whether or not the **dgHeaderControl** property of the control that was clicked on (**the target**) is empty (2). If the property is not empty then you know that the user clicked on a column header.

Once you know that the user clicked on a column header you can display the contextual menu. In this example the name of the column that was clicked on (3) is appended to "Popup" in order to get the name (*State Popup*) of the popup button to display (4).

Copy & Paste

```
on mouseDown pBtnNum
  ## Let Data Grid process mouseDown
  dgMouseDown pBtnNum
  if pBtnNum is 3 then
    if the dgHeaderControl of the target is not empty then
      ## the user clicked on a column header
      ## Get column name
      put the dgColumn of the target into theColumn
      ## Get name of contextual button to use
      put theColumn && "Popup" into theContextualButtonName
      ## Display contextual if exists
      if there is a button theContextualButtonName then
        popup button theContextualButtonName
      end if
    end if
  end if
end mouseDown
```

The Result

Now when a **right-click** occurs on the State column the *State Popup* popup button is displayed as a contextual menu.

What if I Need to Work With `htmlText` or `rtfText`?

By default all table columns will have their **text** property assigned. If you need to work with `htmlText` or `rtfText` then you need to provide your own behavior for the table columns.

Option 1: Set the `dgProp["Default Column Behavior"]` Property

The simplest way to change the default behavior of table columns is to set the **`dgProp["default column behavior"]`** property of the data grid. You set this property to a **button id** whose script contains the behavior you would like the columns to have. Doing this overrides the default behavior provided by a button on the **revDataGridLibrary** stack.

```
set the dgProp["default column behavior"] of group "DataGrid" to the long id\
  of button "My Custom Column Behavior" of stack "MyStack"
```

Here is an example of a **FillInData** command that takes a column value encoded as UTF-8 and sets the **unicodetext** of the column field.

Important: If you decide to write your own default behaviour script make sure to include the **dgDataControl** `getProp` handler. This is required in order for the data grid to work properly.

```
getprop dgDataControl
  -- Required by library so that it can locate the control.
  return the long ID of me
end dgDataControl
```

To see the default column script the data grid uses enter the following in the **Message box**:

```
edit script of button "Default Column" of stack "revDataGridLibrary"
```

Option 2: Create a Custom Column Template

Alternatively you can create your own custom column template for a particular column. This allows you complete control over the look and feel of the column. See section [How Do I Customize A Table's Columns?](#)

How Do I Display Line Numbers in a Table?

This lesson will show you how to dynamically display line numbers in your data grid table.

The Table

Here is the table to display line numbers in.

Add Line Number Column

Select the data grid and open its **Property Inspector**. Select the **Columns** pane (1) and add a column using the "+" button (2). Rename the column **Line Number** (3).

Reorder and Create Column Template

Use the up arrow (1) to move the **Line Number** column to the top of the list.

Create a template for this column by clicking on the "+" button at the bottom of the **Property Inspector** (2).

Edit Line Number Group

After creating the column template the card with the template group will open. As you can see a group called **Line Number**, corresponding to the column's name, has been created and within that group is a single field object.

We do not need to actually do anything to the template so simply **close** and **save** it when prompted.

Edit Column Behavior

Now all that is left is to define the behavior for this column. Go back to the **Columns** pane of the data grid's **Property Inspector** and click the **Column Behavior...** button (1) to open its script editor.

Edit Behavior Script

There isn't much to do here. In **FillInData** you just want to (1):

```
set the text of field 1 of me to the dgLine of me .
```

The **dgLine** is a property of a **row template** and this will display the line number for the current line. In **LayoutControl** you just need to ensure the rect of the field fills the entire cell (2).

Refresh Data Grid

In the **Property Inspector** click the Refresh Data Grid button.

The Result

Now the line numbers appear in the table.

But How Do I Set the dgText Then?

You may be wondering how you would set the **dgText** property of this data grid since we have added a **Line Number** column as the first column in the table.

When you set the **dgText** of a data grid you can pass a parameter specifying whether or not the first line of text contains the names of the columns that the data should map to. Just pass in true and prepend the column names to your data like this:

```
put "State" & tab & "State Code" into theColNames
set the dgText [true] of group "Data Grid" to theColNames & cr & theData
```

How Do I Customize Column Sorting?

This lesson will demonstrate how to customize sorting for columns by handling the **SortDataGridColumn** message that is sent when the user clicks on a column.

A Data Grid Table

The goal is to customize when happens when this table is sorted by the **Line Number** column.

The Data Grid Group Script

A Data Grid calls a built in handler **SortDataGridColumn** whenever the **dgProps ["sort by column"]** property is set. This includes when the user clicks on a column header to sort that column. By placing a modified version of the **SortDataGridColumn** in the data grid's script this message can be captured and the behavior modified.

Place the following in the script for the data grid.

Copy & Paste

```
on SortDataGridColumn pColumn
  switch pColumn
  case "Line Number"
    ## ReverseSort is a helper provided by the Data Grid.
    ## It merely reverses the current sort.
    ReverseSort pColumn ## pass in column so header is hilited
    break
  case "State Code"
    ## Just provided as an example of how to prevent the user
    ## sorting on a column. In this case the "State Code" column.
    ## We accomplish this by NOT passing the message.
    break
  default
    ## For all other cases, pass the message so that the data grid
    ## will be sorted based on column settings.
    pass sortDataGridColumn
  end switch
end SortDataGridColumn
```

The way the custom **SortDataGridColumn** handler works is simple. Passing the message will cause the data grid to use its default column sorting routines. (1) Set this as the **Default** case.

Not passing the message prevents the data grid from performing it own sorting (2) and (3).

In this example, clicking on the **Line Number** column will just reverse whatever current sort exists. This is done by calling **ReverseSort**, a built-in data grid helper that reverses whatever the current sort order

is. (2)

The Result

Clicking on the **Line Number** column (1) now reverses the sort.

Custom Sort Example

Here is another example of a custom sort. Try it out and see if you can determine what it does.

```
## Place in Data Grid script.
## This example shows how to perform a custom sort of data. This is similar
## to the internal routine that the data grid uses.
on SortDataGridColumn pColumn
  ## Begin by building a list of lines with two items (tab delimited)
  ## The first item is the index of each record stored in the data grid.
  ## The second item is the key you are going to sort on.
  put the dgData of me into theDataA
  repeat for each key theIndex in theDataA
    put theIndex & tab & theDataA[theIndex][pColumn] & cr after theData
  end repeat
  delete the last char of theData
  set the itemdelimiter to tab
  ## perform sort. Lookup the current sort direction for this column
  ## and use that as sort direction
  put the dgColumnSortDirection[pColumn] of me into theSortDirection
  if theSortDirection is "ascending" then
    sort lines of theData ascending by item 2 to -1 of each #
  else
    sort lines of theData descending by item 2 to -1 of each
  end if
  ## Rebuild the order of indexes in the data grid
  put empty into theIndexSequencing
  repeat for each line theLine in theData
    put item 1 of theLine & comma after theIndexSequencing
  end repeat
  delete the last char of theIndexSequencing
  ## Set the dgIndexes property to new order
  set the dgIndexes of me to theIndexSequencing
  ## Tell data grid to hilite column
  HiliteAndStoreSortByColumn pColumn
end SortDataGridColumn
```

How Do I Disable Column Sorting?

This lesson will show you how to disable column sorting in a Data Grid table.

Sort By No Column

Notice in our example, the **State** column header is highlighted and there is a sorting arrow indicating an ascending sort.

To begin, set the **sort by column** property to empty. This will remove any column sorting that might be active.

```
set the dgProp["sort by column"] of group "DataGrid 1" to empty
```

Just copy and paste the above into the **Message box** to execute it.

Your table headers should now appear the same, with no highlights or sorting arrow.

Edit Data Grid Script

Edit the Data Grid group script by **right clicking** on the Data Grid and selecting **Edit Script**.

Add SortDataGridColumn Handler

When the user clicks on a table column header the **SortDataGridColumn** message is sent to the Data Grid. Intercepting this message and not passing it effectively disables sorting. For more information about how **SortDataGridColumn** works see [How Do I Customize Column Sorting?](#).

Add the following code to your Data Grid group script and compile.

Copy & Paste

```
on SortDataGridColumn pColumn  
end SortDataGridColumn
```

Now when you click on the table header the data will not be sorted.

Note: Once you add the **SortDataGridColumn** handler to a Data Grid setting the **dgProp["sort by column"]** property will no longer do anything. If you need to set the **sort by column** property later on you would need to remove the **SortDataGridColumn** handler from the script, or at least **pass** the message.

How Do I Perform an Action After the User Sorts a Data Grid?

This lesson will show you how to modify a data grid script so that you can perform an action whenever a data grid is sorted.

Add SortDataGridColumn To Data Grid Group Script

You can customize what happens when a data grid is sort by adding a **SortDataGridColumn** handler to your data grid's script. For more information about howSortDataGridColumn works please see the lesson [How Do I Customize Column Sorting?](#).

In order to perform an action AFTER the sort has been applied you simply need to call the handler the Data Grid behavior uses to sort but NOT pass the **SortDataGridColumn** handler. Doing so sorts the data just like the Data Grid would and then perform any additional actions you would like.

```
## Place in Data Grid script.
on SortDataGridColumn pColumn
## Call Data Grid sort routine by hand
SortByColumn pColumn
## Now data has been sorted, do whatever you need to do.
-- my code
.....
.....

## DON'T PASS the "SortDataGridColumn" message!
end SortDataGridColumn
```

How Do I Align Decimals in a Column?

This lesson will show you how to create a custom column template that provides decimal alignment. You can [download the stack](#) used to create this lesson.

Create a Data Grid Table

Add Some Data

Here a couple of rows of decimal numbers have been added to experiment with.

Create Column Template

Select the **Columns** pane (1) in the **Property Inspector** and click the **Add custom column behavior** button (2). This will create a custom column template and behavior script.

Notice how the **Column Behavior...** button has become enabled.

Edit Column Template

When you create a custom template for a column a new group, named after the column, is added to the data grid's **row template**.

Here you see the **Project Browser** with the template for the Data Grid expanded showing the group for this column (*Col 1*) with a field `_ColumnData_` to hold the column's data.

Edit Column Template Group

Double-click on the column group in the **Project Browser** (1). This will select it on the template card (2) and then click the **Edit Group** button (3) to edit the contents of the column group.

Be sure to double-click on the **group** icon and not the group's name. Double-clicking on the object's name will simply allow you to edit the name, which is not what is wanted.

Create 3 Fields

A column that aligns decimals in the center of the column, and is editable if the user double-clicks on the cell, is achieved by creating three fields.

Since the column template group already had a field named `_ColumnData_` double-click this name (in the **Project Browser**) and rename it to *Leftvalue* (1). Open the **Property Inspector** for this field and set its **textalign** to **right**. **Duplicate** the *Leftvalue* field and name it *Rightvalue* (2). Set the its **textalign** to **right** too. Finally **Duplicate** this field and **rename** the duplicate to **ForEditing** (3). The text of this field should be empty. Make sure that the **ForEditing** field has a higher layer than *Leftvalue* or *Rightvalue*.

Don't worry too much about positioning as the fields will be resized and repositioned when drawn in the data grid.

Click on the card background of the editing template and then the **Stop Editing** button and **close** and **save** the window.

Edit Column Behavior

Now that the objects we need to display the column data have been created its time to tell the column what to do when it is drawn. Return to the **Property Inspector** for the data grid and from the **Columns** pane (1) click the **Column Behavior** button (2). This will open the behavior script for the column.

Update FillInData

The decimal number is going to be displayed by assigning the left side of the decimal to the *Leftvalue* field and the right side of the decimal to the *Rightvalue* field.

```
set the itemdelimiter to "."
set the text of field "Leftvalue" of me to max(0, item 1 of pData)&"."
set the text of field "Rightvalue" of me to max(0, item 2 of pData)
```

Update LayoutControl

The left and right fields are now resized so that each takes up half of the available width to align the decimal in the center of the cell(1). Finally we resize the **ForEditing** field to take up the entire cell width(2). This is the field that the user can double-click on to edit the cell contents.

```
put the rect of field "Leftvalue" of me into theRect
put item 1 of pControlRect into item 1 of theRect
put item 1 of pControlRect + \
    round((item 3 of pcontrolRect - item 1 of pControlRect) / 2) \
    into item 3 of theRect
set the rect of field "Leftvalue" of me to theRect

put the rect of field "Rightvalue" of me into theRect2
put item 3 of theRect into item 1 of theRect2
put item 3 of pControlRect into item 3 of theRect2
set the rect of field "Rightvalue" of me to theRect2

set the rect of field "ForEditing" of me to pControlRect
```

Update EditValue

EditValue is the handler that is called by the data grid to edit a cell value. Usually this is because the user double-clicked on the cell or tabbed into the cell.

EditFieldText displays, by default, the **htmltext** of the field you are editing. This default can be overridden by setting the **dgTemplateFieldEditor[text|htmltext|rtftext|unicodetext|utf8text]** of the data grid. Here the **dgTemplateFieldEditor["text"]** of the data grid is set to **text(1)**. Of course the *ForEditing* field has no text in it by default and the "left" and "right" field values need to be combined as documented in the handler.

Also, if the user double-clicks on this cell they actually double-click on the **ForEditing** field (as it has the higher layer) and it is thus *the target* (2)

```

command EditValue
  ## Assign text the the editor will have when it opens
  ## We combine the left and right field values.
  Set the dgTemplateFieldEditor["text"] of the dgControl of me to \
    the text of field "Leftvalue" of me & the text of field "Rightvalue" of me
  ## The "ForEditing" field is what the user actually clicked on.
  ## This is the "target"
  EditFieldText the long ID of field 1 of the target, the \
    dgIndex of me, the dgColumn of me
end EditValue

```

Refresh Data Grid

Now that the necessary changes have been made refresh the data grid so that it redraws with the new template controls. From the **Basic Properties** pane of the inspector palette click the **Refresh Data Grid** button. Alternatively you can send **ResetList** to the data grid.

```
send "resetList" to group "datagrid 1"
```

Test

The decimals all appear in the center of the cell. Double-clicking brings up an editing field as shown here. Finally the decimal remains in the center of the cell if the column is resized.

How Can I Colorize Individual Lines in a Table?

This lesson will show you how to change the color of individual lines in a Data Grid table. Before you begin this lesson you will need to read the lesson [How Do I Override the Default Behavior for Rendering Data to a Cell?](#)

Attached Files

[Color_Individual_Line_in_Table.rev](#)

Coloring a Single Line

In this example we are going to modify the basic example created in the lesson mentioned above so that it colors a row red if the row's **line has error property** is **true**.

By clicking on the **Toggle Line 3 Color** button the color of line 3 turns red. The code in the button extracts the data for line 3 from the data grid, toggles the **line has error** property and reassigns the new data to line 3.

```
on mouseUp pMouseBtnNo
    put the dgDataOfLine[3] of group "DataGrid 1" into theDataA
    put not theDataA["line has error"] into theDataA["line has error"]
    set the dgDataOfLine[3] of group "DataGrid 1" to theDataA
end mouseUp
```

Now let's look at what the default custom column behavior looks like for this data grid.

The Default Custom Column Behavior

Here is the relevant code from the default custom column behavior for this data grid. The key addition to the code is the **SetForegroundColor** handler. This handler checks the value of the **line has error** column for this row (using **GetDataOfIndex**) and if it is true then changes the cell to red. Since each cell in this row has the same value for **line has error** the entire row appears red.

```
on FillInData pData
    -- This message is sent when the Data Grid needs to populate
    -- this template with the column data. pData is the value to be displayed.
    set the text of me to pData
    SetForegroundColor
end FillInData

on LayoutControl pControlRect
    -- A default column is just a field.
end LayoutControl

setprop dgHilite pBoolean
    -- This custom property is set when the highlight of your column template has
    -- changed. You only add script here if you want to customize the highlight.
    if pBoolean then
        set the foregroundcolor of me to the dgProp["hilited text color"] of the
        dgControl of me
    else
        SetForegroundColor
    end if
end dgHilite

private command SetForegroundColor
    if GetDataOfIndex(the dgIndex of me, "line has error") then
        set the textcolor of me to red
    else
        set the textcolor of me to black
    end if
end SetForegroundColor
```

Working With Data Grid Forms

How Do I Create a Form With Variable Line Heights?

This lesson will show you how to create a data grid form with variable height lines by modifying the default data grid **Row Template** and **Row Behavior**.

Turn Off "Empty row height"



To begin, turn off **Empty row height** for your data grid form in the **Property Inspector**.

Edit Row Template



Open the card that has the row template group by clicking the **Row Template** button.

Turn Off dontWrap Property



The default row template for a data grid has a single field that displays data. With **Select Grouped** turned on (1) select the field (2). Using the **Property Inspector** turn off **dontWrap** (3). You can now close the stack.

Edit Row Behavior



Return to the Property Inspector for your data grid. Edit the **Row Behavior** by clicking on **Edit Script** button.

Script Field to Resize to Fit Height

Only a few modifications are needed to make to the default **LayoutControl** handler resize the field to fit the height.



The default **LayoutControl** handler resizes the field to fill the available width (1). After that is done you then need to resize the field to fill the **formattedHeight** (2). Finally, adjust the rect of the Background graphic to take into account the new field height (3).

```

on LayoutControl pControlRect
  local theFieldRect
  ## Expand the field to fill the available width
  put the rect of field "Label" of me into theFieldRect
  put item 3 of pControlRect - 5 into item 3 of theFieldRect
  set the rect of field "Label" of me to theFieldRect

  ## Now resize the field to fit its content
  put item 2 of theFieldRect \
    + the formattedheight of field "Label" of me - \
    the bottommargin of field "Label" of me \
    into item 4 of theFieldRect
  set the rect of field "Label" of me to theFieldRect

  ## Now update the bounding rect to match total the height
  ## you want this row to have (note there is an additional 4
  ## pixels added to the bottom
  put item 4 of theFieldRect + 4 into item 4 of pControlRect

  set the rect of graphic "Background" of me to pControlRect
end LayoutControl

```

Refresh Data Grid Contents

You can now refresh the data grid contents to see the new behavior.

Notice how each line is resized to fit all of the text.

How Do I Sort Records by a Specific Key's Values?

You can sort the rows of a data grid form using the **SortDataByKey** command.

Let's look at an example.

Example

This card has a data grid and an option menu. The option menu contains three values that you can sort by: *First Name*, *Last Name* and *Title*.

Option Menu Code

The code to perform the sort is pretty straight forward.

1. Determine which array key of the data grid form to sort by. This will be one the keys created when

the data was assigned to the **dgData**. If you used the **dgText** property then the key will be "Label 1" or "Label 2", etc.

2. Determine the sort type, direction and whether or not the sort is case sensitive.
3. Call the **SortDataByKey** command.

Copy & Paste

```
on menuPick pItemName
  switch pItemName
    case "First Name"
      put "FirstName" into theKey
      break
    case "Last Name"
      put "LastName" into theKey
      break
    case "Title"
      put "Title" into theKey
      break
  end switch

  put "international" into theSortType
  put "ascending" into theDirection
  put "false" into isCaseSensitive
  dispatch "SortDataByKey" to group "Datagrid 1" with \
    theKey, theSortType, theDirection, isCaseSensitive
end menuPick
```

The Result

Here is what the sort looks like after selecting *First Name* and after selecting *Last Name*.

How Do I Create Rows That Can Expand/Contract?

This lesson will show you how to make a Data Grid form with rows that expand and contract when clicking on an arrow.

Attached Files

[ExpandingRow.rev](#)

Expanding and Contracting Rows

This is what the example stack looks like. By default all of the rows are contracted and only show names.

Clicking on the arrow next to a name expands the row to show the description of the person.

Setting the Data Grid Properties

To create a Data Grid form with rows that expand/contract you need to turn off **fixed control height** in the data grid's **Property Inspector**.

Note: To turn off the property using script you set the **dgProp["fixed row height"]** property to **false**.

Updating the Row Behavior Script

In order to get expanding rows working a couple of things need to be taken into account in the **Row Behavior Script**.

The **FillInData** handler should show or hide controls as necessary. The example stack stores the expanded state of each row in the **expanded** key.

In the **LayoutControl** handler the position of the controls and background graphic based on whether or not the row is expanded or contracted are adjusted . Since the **fixed row height** property is set to **false** the Data Grid computes the total height of the data based on the height of each row after calling **LayoutControl**.

When the user clicks on the arrow the example stack updates the **expanded** key in the row's data and then redraws the row. Using **SetDataOfIndex** in conjunction with **RefreshIndex** is the most efficient way to do this.

pdataArray["expanded"] is a boolean which determines the state of the row.

the short name of the target is "ArrowContracted"

is an expression that evaluates to **true** or **false** depending on the target. If the *ArrowContracted* graphic is visible and thus clicked on, the expression evaluates to **true** which in turn sets the row to be in its **expanded** state as per the **FillInData** handler.

Thus this little snippet effectively toggles the state of the row.

How Can I Speed Up Drawing When "Fixed Row Height" Is False?

When you set the **dgProp["fixed row height"]** property of a Data Grid to **false** the Data Grid must draw all records in order to determine the total height. That means that **FillInData** and **LayoutControl** are called for every record in the Data Grid. This can be time intensive depending on the amount of data.

This lesson will show you a technique that can speed up the calculation of the total height of the data in

situations where the rows can only have heights of *known* values.

CalculateFormattedHeight

When the Data Grid loops through the data to calculate the height the message **CalculateFormattedHeight** will be sent to the **Row Template**. If your Row Behavior handles this message then the Data Grid will use the integer value that you return rather than calling **FillInData** and **LayoutControl**.

Here is an example that uses **CalculateFormattedHeight** to return the height of a row based on whether or not the row is expanded. Just place the **CalculateFormattedHeight** handler in your Row Behavior script.

```
on CalculateFormattedHeight pdataArray
  if pdataArray["expanded"] then
    return 74
  else
    return 20
  end if
end CalculateFormattedHeight

on FillInData pdataArray
  ...
end FillInData
```

Using the Built-in Field Editor

How Do I Open a Table Cell for Editing?

By default a table cell can be edited if the user double-clicks on the cell. This tutorial explains what goes on in the default column behavior so you can customize behavior if you would like.

Note: The default column behavior is stored in button "Default Column" of stack **"revDataGridLibrary"**. See [How Do I Override the Default Behavior for Rendering Data to a Cell?](#)

What You Need to Know

In order to edit the columns of a data grid table you need to know about the following:

1. The **EditFieldText** command
2. The **EditValue** message
3. The **EditKey** and **EditKeyOfIndex** commands
4. The **CloseFieldEditor** command which can be sent as a result of calling **EditFieldText**.

Read up on the entries for **EditFieldText**, **EditValue**, **EditCell/EditCellOfIndex** in the API section of the Dictionary.

EditFieldText

The **EditFieldText** command will create an editor for a field that you specify. The default column behavior calls this command with three parameters so that data is automatically saved after the user finishes editing.

EditValue

EditValue is the message that is sent to a row when a request to edit a field's contents has been made. The default column behavior calls **EditFieldText** when this message is received.

EditCell and EditCellOfIndex

There are two commands that will open a cell for editing. They are **EditCell** and **EditCellOfIndex**. Each takes the name of the column you want to edit and the line or index you want to edit. Here are two example scripts showing how you could use these commands in the script of a data grid.

```
put "FirstName" into theColumn
put the dgHilitedLine of me into theLineNo
EditCell theColumn, theLineNo

put "FirstName" into theColumn
put the dgHilitedIndex of me into theIndex
EditCellOfIndex theColumn, theIndex
```

Either of the above calls will trigger the **EditValue** message.

```
on EditValue
  ## Example of opening a field editor for the targeted column.
  ## Since I'm passing in parameters 2 and 3 any changes will
  ## automatically be saved to the dgData.
  EditFieldText the long id of me, the dgIndex of me, the dgColumn of me
end EditValue
```

CloseFieldEditor

If the user changes any content in the field editor this message will be sent to the field targeted in the first parameter sent to **EditFieldText**. Read the API docs for **EditFieldText** which discusses this message.

Example of storing value in **dgData** in **CloseFieldEditor**. This would be required if you only passed in one parameter to **EditFieldText**.

```
on CloseFieldEditor pFieldEditor
  put the dgIndex of me into theIndex
  put the dgDataOfIndex[theIndex] of the dgControl of me into theDataA
  put the text of pFieldEditor into theDataA[the dgColumn of me]
  set the dgDataOfIndex[theIndex] of the dgControl of me to theDataA
end CloseFieldEditor
```

How Can the User Edit Field Content in a Data Grid Form?

The data grid commands for creating an editor for a particular field in a row template. This lesson will show you how to use them.

What You Need to Know

In order to edit field contents in a data grid form you need to know about the following:

1. The **EditFieldText** command
2. The **EditValue** message
3. The **EditKey** and **EditKeyOfIndex** commands

Read up on the entries for **EditFieldText**, **EditValue** and **EditKey/EditKeyOfIndex** in the API documentation.

EditFieldText

The **EditFieldText** command will create an editor for a field that you specify. You can use this command to create a field editor for a field in your row template.

EditValue

EditValue is the message that is sent to a row when a request to edit a fields contents has been made. You can call **EditFieldText** from within a handler for this message to begin an editing operation.

EditKey and EditKeyOfIndex

EditKey and **EditKeyOfIndex** will trigger the **EditValue** message in a row. Each takes the name of the key you want to edit and the line or index you want to edit. Here are two example scripts showing how you could use these commands in the script of a data grid.

Placed in script of row template behavior

```
on mouseDown pBtnNum
  if pBtnNum is 1 then
    ## Did the user click on the FirstName field?
    if the short name of the target is "FirstName" then
      put "FirstName" into theKey
      put the dgHilitedLine of me into theLineNo
      EditKey theKey, theLineNo
    end if
  end if
end mouseDown

on mouseDown pBtnNum
  if pBtnNum is 1 then
    ## Did the user click on the FirstName field?
    if the short name of the target is "FirstName" then
      put "FirstName" into theKey
      put the dgHilitedIndex of me into theIndex
      EditKeyOfIndex theKey, theIndex
    end if
  end if
end mouseDown
```

Either of the above calls will trigger the **EditValue** message. The **EditValue** can be thought of as a central message where you can open a field for editing text. A handler for this message is where you will call **EditFieldText**.

```

on EditValue pKey
  ## Example of opening a field editor for the field displaying
  ## the value for pKey
  ## Since we are passing in parameters 2 and 3 any changes
  ## will automatically be saved to the dgData.
  ## 'me' is the Data Grid in this case.
  EditFieldText the long id of field pKey of me, the \
    dgHilitedIndex of me, pKey
end EditValue

```

CloseFieldEditor

If the user changes any content in the field editor this message will be sent to the field targeted in the first parameter sent to **EditFieldText**. Read the API docs for **EditFieldText** which discusses this message.

Here is an example of storing the new value in the **dgData** of the Data Grid in the **CloseFieldEditor** handler. Manually storing the value would be required if you only passed in one parameter to **EditFieldText**.

This example script would be in the **Row Behavior** script as it uses the **dgIndex of me** property.

```

on CloseFieldEditor pFieldEditor
  ## 'me' is the row control
  put the dgIndex of me into theIndex
  put the dgDataOfIndex[theIndex] of the dgControl of me into theDataA
  put the text of pFieldEditor into theDataA[the dgColumn] of me
  set the dgDataOfIndex[theIndex] of the dgControl of me to theDataA
end CloseFieldEditor

```

How Can I Edit the Text as UTF-8, UTF-16 or HTML?

The default Data Grid behavior when editing cell contents is to use the **text** property of the cell as the default value to be edited. This lesson will show you how to provide a specific value for the Data Grid to use as the value to edit.

The technique described requires that you create a custom column behavior as outlined in the lesson [How Do I Override the Default Behavior For Rendering Data to a Cell](#) .

The Default Behavior

Assume you have a column in your table that displays text with some styling. In this example the last name in column 2 for **some** entries is italic.

We can accomplish this by using the formatting information contained in the data.

However, when you edit the content of the cell the formatting is lost. This is because the Data Grid edits the **text** property of the field by default.

Changing the Default Value For the Editor

You can change the default value of that the field editor uses by setting a value of the **dgTemplateFieldEditor** property of the Data Grid. You set the property before calling **EditFieldText**.

For example, if you wanted to display the **italic** text for editing you could set the `dgTemplateFieldEditor["htmltext"]` property to the `htmltext` of the column being edited:

```
set the dgTemplateFieldEditor["htmltext"] of the dgControl of me to the htmltext of me
```

Other properties you can set include **rtftext**, **text**, **unicodetext** and **utf8text**.

The Result

Here is the result of using the example code above in the custom column behavior. Notice how the text being edited is bold.

How Can I Select the Text in the Edit Field When It Opens?

The default Data Grid behavior when editing cell contents is to put the cursor at the end of the field. This lesson will show how to tell the Data Grid that all of the cell text should be selected.

The technique described requires a custom column behavior as outlined in the lesson [How Do I Override the Default Behavior For Rendering Data to a Cell](#) .

The Default Behavior

By default the Data Grid will place the insertion point at the end of the text of a cell when you start editing a value.

Selecting The Text

You can tell the Data Grid to select all of the text by setting a special custom property called **dgTemplateFieldEditor**. If you set the **dgTemplateFieldEditor["select text"]** of the Data Grid to **true** then the data Grid will select the cell text.

An Example Script

```
## Select all text before opening
set the dgTemplateFieldEditor["select text"] of the dgControl of me to true
```

Add this line of text to the **EditValue** handler in a custom column behavior. Just place the code right before the call to **EditFieldText**.

How Do I Save Changes the User Makes in an Editor Field to an External Data Source?

When Calling EditFieldText With 3 Parameters (Simpler)

When calling **EditFieldText** with all three parameters (which is what a data grid column does by default) the data grid will automatically save the text that the user enters in the **dgData** array. That means that all you need to do is save the text of the editor field to your external data source in the **CloseFieldEditor** handler. Note that if you save anything other than the text of **pFieldEditor** then your data source will not match the **dgData** value.

Here is an example script that goes in the Data Grid group script.

```
## An example that saves
on CloseFieldEditor pFieldEditor
  put the dgColumn of the target into theColumnBeingEdited
  put the text of pFieldEditor into theNewText
  ## Save data to database using command I defined
  put "Person" into theTable
  ## Get the unique id of the row in the database we want to edit
  put GetDataOfIndex(the dgIndex of the target, "id") into theRowID
  SaveDataToDatabase theTable, theRowID, theColumnBeingEdited, theNewText
end CloseFieldEditor
```

When Calling EditFieldText With 1 Parameter (More Flexible)

If you are working with data grid forms or decide to override the default behavior for data grid columns then you will make the call to **EditFieldText** yourself. For complete control over what gets saved you only pass in 1 parameter to **EditFieldText**. By passing in 1 parameter you are responsible for saving any changes to **dgData**.

Assume you made the following call in a data grid form:

```
EditFieldText the long id of field "Name" of me
```

A **CloseFieldEditor** message would still be sent when the user changes the contents of the field but would need to contain code for saving the changes to the **dgData** array.

Here is an example script that goes in the Data Grid group script.

```

## CloseFieldEditor placed in data grid script
on CloseFieldEditor pFieldEditor
  put the dgColumn of the target into theColumnBeingEdited
  ## Store UTF8 text
  put unicode(the unicodetext of pFieldEditor, "UTF8") into theNewText
  ## Save data to database using command I defined
  put "Person" into theTable
  ## Get the unique id of the row in the database we want to edit
  put GetDataOfIndex(the dgIndex of the target, "id") into theRowID
  SaveDataToDatabase theTable, theRowID, theColumnBeingEdited, theNewText
  ## Update dgData.
  ## Setting dgDataOfIndex will refresh the data grid display as well as update
  dgData
  put the dgDataOfIndex[ the dgIndex of the target] of me into theDataA
  put theNewText into theDataA[theColumnBeingEdited]
  set the dgDataOfIndex[the dgIndex of the target] of me to theDataA
end CloseFieldEditor

```

How Can I Customize the Field Editor Behavior?

By default the Data Grid field editor allows users to enter data and save it back to the Data Grid. If you need data entry to behave differently you can assign your own behavior script to the field editor before it opens. This lesson will show you how.

Create Your Behavior Script

1. Create a button (1) to hold the behavior script you want to use with the Data Grid field editor. Here it is placed on the same card as the data grid.
2. Set the script of the button (2) to the script of button "Field Editor" of stack "revDataGridLibrary". This is the behavior script that Data Grid uses by default and is a good place to start when customizing the behavior.

Customize Your Script

Open the button's script...

Make any customizations you need to make.

Assign Your Custom Behavior to Field Editor

Whenever the Data Grid displays the field editor (e.g. the user double-clicks on a cell in a table) a **preOpenFieldEditor** message is sent to the Data Grid. The first parameter is a reference to the field editor control. This is where a behavior script can be assigned to the field.

Building Standalones With the Data Grid

What Do I Need to Do to Deploy a Standalone With a Data Grid?

A data grid relies on the stack **revDataGridLibrary** in order to function properly. This lesson will describe how to include this stack in your standalone applications.

How The Standalone Builder Adds The Necessary Files

When you add a data grid to a stack, a substack is created whose name begins with "Data Grid Templates".

When you build a standalone application LiveCode uses the settings you have set for your application. These are on the **General** tab of the **Standalone Applications Settings**. By default LiveCode will look for a substack whose name begins with "Data Grid Templates". If it finds one then the **revDataGridLibrary** stack is added to your standalone application.

But What About Launcher Stacks (Splash Stacks)?

Some developers prefer to use a launcher (or splash) stack technique. This technique builds a standalone using a stack with very little code in it that in turn opens the main stack that would contain the data grid templates. In this case the stack used to build the standalone will not have a substack whose name begins with "Data Grid Templates" as no data grids have been added to it.

In this case you will need to tell LiveCode that you require the inclusion of the Data Grid Library. As of version LC 8, the DataGrid library **DataGrid** has been added to the list of **Script Libraries** in the **Standalone Settings** dialog. You can now force the inclusion of the **DataGrid** library, to ensure that any stack loaded by the standalone can use Data Grids.

Open the **General** tab of the **Standalone Applications Settings** (1) and select **Select inclusions for the standalone application** (2)

Now open the **Inclusions** tab (1), locate the **DataGrid** library, and then click the checkbox (2) to include it.

Useful Things to Know

What Sorts of Things Should I Not Do in Order to Avoid Needless

Suffering?

There are some things that you could do that will cause you to scratch your head when things go wrong. Since knowing is half the battle we will share the issues we are aware of.

Don't Call a Handler That Redraws the Data Grid From Within a Control in the Data Grid

This will generate an error since you are deleting a control that is currently executing code. This is a no-no and the LiveCode engine will complain and stop executing. You can avoid calling a handler that refreshes the data grid from within a control by a) using **send in time** or b) placing the code in the data grid script itself.

Example A

```
## Script that is in the row behavior for a data grid.
send "DeleteIndex theIndex" to the dgControl of me in 0 seconds
```

Example B

```
## Script that deletes hilited index on mouseUp.
## Place in the data grid script.
on mouseUp pMouseBtnNum
  if pMouseBtnNum is 1 then
    put the dgHilitedIndex of me into theIndex
    DeleteIndex theIndex
  end if
end mouseUp
```

Don't try to draw a Data Grid on a card that is not open

When a Data Grid renders, it dynamically creates fields and accesses certain properties. Some of these properties can not be properly reported by the LiveCode engine unless the field is on an open card.

Do not lock messages when accessing data grid properties.

If messages are **locked** when you try to access a data grid property, for example

```
set the dgProps["alternate row color"]** of group "DataGrid"
```

the correct value will not be returned/set. A Data Grid relies on getProp/setProp handlers to function. When messages are locked these are not triggered.

Do not password protect the Data Grid Templates Stack

The data grid copies the templates from the **Data Grid Templates xxxx** stack. If you password protect this stack then the data grid will be unable to copy the templates.

Don't Rename the "Data Grid Templates" Stack

If you rename this stack then all of your data grids with templates stored in the stack will stop working. Since the data grid can no longer locate the custom templates they will fail to draw properly.

Don't Try to Search When Data is Being Loaded From an External Source

Stop Editing the Template Group Before Drawing Your Data Grid

When you **edit** a group in LiveCode the engine no longer knows that the group exists. If you try to draw a data grid while editing its template group then the data grid will fail to draw. Make sure you have selected **Stop Editing** from the **Object** menu or clicked on the **Stop Editing** icon on the **Toolbar**.

Advanced Options

Displaying Large Amounts of Data

Setting the **dgText** property or creating an array and setting the **dgData** property of a data grid is the easiest way to display your data. But what about situations where creating an array is too time intensive and you already have the data in another format? A data grid can handle these situations for you as well using a feature called callbacks.

Download the attached sample stack and database that shows how to use the techniques described in this lesson to display data from a SQLite database.

Attached Files

[datagrid_databases.zip](#)

The dgNumberOfRecords Property

Normally a data grid reports the number of records based on the number of numeric indexes in the first dimension of the **dgData** array. If you set the **dgNumberOfRecords** property, however, the data grid stops using an internal array and issues a callback message whenever it needs to display data in a line.

Important: When using this technique properties like **dgData**, **dgText**, **dgDataOfIndex**, etc. will no longer return values. The data grid is just displaying records from your data source. It does not store any of that data internally.

GetDataForLine Callback

This diagram demonstrates how this works. If you were to set the **dgNumberOfRecords** to 10000 then the data grid would start sending the **GetDataForLine** message whenever it needed to display a row. Your responsibility is to fill in the data that the data grid needs by handling the **GetDataForLine** message. To do that you define **GetDataForLine** as follows:

```
command GetDataForLine pLine, @pDataA
end GetDataForLine
```

You can define this handler in the data grid script or anywhere else in the message path. Just fill in pDataA with the appropriate data and the data grid will display it.

Creating a Data Grid by Hand

This lesson will show you how to create a data grid through script.

Copy Data Grid From revDataGridLibrary

The data grid template is stored in the **revDataGridLibrary** stack and can be copied using some code similar to this:

```
copy group "DataGrid" of group "Templates" of stack "revDataGridLibrary" \
  to card "MyCard" of stack "MyStack"
put it into theDataGridRef
```

Set The "style" Property

```
set the dgProp["style"] of theDataGridRef to "table" or "form"
```

Assign a Row Template

```
set the dgProp["row template"] of theDataGridRef \
  to the long id of group "MyRowTemplate" of stack "MyStack"
```

You should probably create this ahead of time using the IDE. For example, you could create a data grid and then delete it while leaving the row template behind (it will exist on a card in "Data Grid Templates xxx" stack).

How Do I Create a Datagrid Dynamically From Tab Delimited Data

Most of the lessons show how to use and change previously created data grids, and assume you want to create them in the IDE. This lesson will explain how to create a table data grid by script.

Prepare Your Data

To make sure that the data grid can accept your tabular data, you should prepare it with these points in mind:

This example works with tab delimited data. The first row of the data should include the names of the columns (this makes it easier to create columns later on) each name should be unique, no empty names are allowed. One way to create tab delimited data is to choose "save as..." in Excel, and then select "text file (tab delimited)". It is simple to load data into Livecode from such a file:

```
answer file ""
put url ("file:" & it) into theData
```

Create a data grid

Each data grid consists of a group with lots of scripts and objects. Luckily, you do not need to recreate all that from scratch. Instead you can simply copy a template data grid from the data grid library. Because this library is necessary for all data grid functionality, it will be added to standalones by the IDE automatically. Please also see [What Do I Need to Do To Deploy a Standalone With A Data Grid?](#) for an explanation when and how you need to add the library to standalones yourself.

The code to copy a datagrid:

```
--copy the template group
copy group "DataGrid" of group "Templates" of stack "revDataGridLibrary" to this
card
set the name of it to "my Datagrid"
set the dgProp["style"] of group "my Datagrid" to "table"
```

Create the Row template

In theory, your data grid is already ready for use. But for the IDE to interact correctly with the new data grid, you also need to add a substack with the row template, just as the IDE would. On the second card of that stack, you will need to add a group called **Row Template**, and you need to point your data grid to that group.

Here is how you could do that:

```
--create row template
put the name of this stack into theMainstack
put "Data Grid Templates" && the seconds into theName
create invisible stack theName
set the mainStack of stack theName to theMainstack
go stack theName
create card
create field "label"
create graphic "Background"
group field 1 and graphic 1
set the name of last group to "Row Template"
go stack theMainstack
--point datagrid to template
set the dgProp["Row Template"] of group "my Datagrid" to the \
  long id of group "Row Template" of stack theName
```

Fill in Your Data

As a last step, you will want to insert the prepared data into the data grid. Again, this assumes that the first row of your data contains an unique description for each column:

```
--add data
put true into firstLineIsNames
set the dgText[firstLineIsNames] of group "my DataGrid" to theData
```

Prepare the Columns of the Data Grid

Because you are creating the datagrid from arbitrary data, you will also need to **dynamically create** all necessary columns. To do that you need to set the **dgProp["columns"]**. Note that it accepts the names only as a return delimited list.

If your data includes the column names in it's first row, you can simply take this and replace tab with return:

```
--create columns  
put line 1 of theData into theColumns  
replace tab with return in theColumns  
set the dgProp["columns"] of group "my DataGrid" to theColumns
```

Everything together

To test out the example scripts, simply copy the following **mouseUp** handler into a button of an empty stack, and click on it with the **browse** tool.

```

on mouseUp
  answer file "tab delimited" with type "any file" or \
    type "tab file|tab|TEXT" or type "txt file|txt|TEXT"
  --in case the user cancels the file dialogue, the script does not proceed
  if it = "" then
    exit mouseUp
  end if
  put url ("file:" & it) into theData

  lock screen --when doing visual stuff, this helps to speed things up
  --make sure the group does not exist already
  if there is a group "my Datagrid" then
    delete group "my Datagrid"
  end if
  --copy the template group
  copy group "DataGrid" of group "Templates" of stack "revDataGridLibrary" \
    to this card
  set the name of it to "my Datagrid"
  set the dgProp["style"] of group "my Datagrid" to "table"
  --position it nicely
  set the rectangle of group "my Datagrid" to 0,the bottom \
    of me + 16,the width of this card, the height of this card

  --create row template
  put the name of this stack into theMainstack
  put "Data Grid Templates" && the seconds into theName
  create invisible stack theName
  set the mainStack of stack theName to theMainstack
  go stack theName
  create card
  create field "label"
  create graphic "Background"
  group field 1 and graphic 1
  set the name of last group to "Row Template"
  go stack theMainstack
  --point datagrid to template
  set the dgProp["Row Template"] of group "my Datagrid" to the \
    long id of group "Row Template" of stack theName

  --create columns
  put line 1 of theData into theColumns
  --no empty column names are allowed, so insert a space when \
    that happens
  replace tab & tab with tab & " " & tab in theColumns
  replace tab with return in theColumns
  set the dgProp["columns"] of group "my DataGrid" to theColumns

  --add data
  put true into firstLineAreNames
  set the dgText[firstLineAreNames] of group "my DataGrid" to theData
end mouseUp

```

Data Grid Tips & Tricks

How Do I Scroll a Row to the Top of the Data Grid Form?

This lesson will show you how to scroll a particular row to the top of the Data Grid form using the **dgRectOfIndex** (or **dgRectOfLine**) property. This technique is useful when your rows are not a fixed height.

Here is selected row in a data grid (1). The goal is to scroll that row to the top of the data grid (2).

How To Do It

Here is the code that will scroll the selected line to the top of the data grid.

Copy & Paste

```
put the dgHilitedIndex of group "DataGrid" into theIndex
put the dgRectOfIndex [theIndex] of group "DataGrid" into theControlRect
put the rect of group "DataGrid" into theGridRect
put item 2 of theGridRect - item 2 of theControlRect into theOffset
set the dgVScroll of group "DataGrid" to the dgVScroll of group "DataGrid" -
theOffset
```

The Result

After executing the code the row will have been moved to the top of the data grid.

How Do I Find Out Which Line the Mouse Is Over?

Each data grid row has a **dgDataControl** property that returns the **long id** of the control. Each control also has a **dgLine** and **dgIndex** property. You can combine the two in order to identify the control the mouse is over.

```
put the dgDataControl of the mousecontrol into theControl
if theControl is not empty then
  put the dgLine of theControl into theLineNo
end if
```

Converting a Database Cursor to a Data Grid Array

This lesson demonstrates a handler that will convert a database cursor into an array that you can use to set the **dgData** property of a data grid.

The Handler

```

command ConvertSQLCursorToArray pCursor, @pOutArrayA
  local i,j
  local theFields
  local theError

  put revDatabaseColumnNames(pCursor) into theFields
  if theFields begins with "revdberr," then
    put item 2 to -1 of theFields into theError
  end if

  if theError is empty then
    put 0 into i
    repeat until revQueryIsAtEnd(pCursor)
      add 1 to i

      repeat for each item theField in theFields
        put revDatabaseColumnNamed(pCursor, theField) into \
          pOutArrayA[i][theField]
      end repeat

      revMoveToNextRecord pCursor
    end repeat
  end if

  return theError
end ConvertSQLCursorToArray

```

The Handler first checks that no errors have been returned. If there are any, the error data is returned.

The handler then steps through the database cursor restructuring the data into an indexed multidimensional array that can be used by the data grid.

How to use

You can call this command as follows.

In this example theCursor is a database cursor that you have opened using revQueryDatabase.

```

local theDataA
ConvertSQLCursorToArray theCursor, theDataA
put the result into theError

```

If theError is empty then dimension 1 of *theDataA* will contain integers from 1 to the number of records in the cursor (**revNumberOfRecords**). Each 1st dimension, in turn, has a key for each column in the cursor (**revDatabaseColumnNames**).

Example

Here is what the array might look like if your cursor had 2 columns (id and name) and 3 rows.

```
theDataA [1] ["id"]
```

```
theDataA [1] ["name"]
```

```
theDataA [2] ["id"]
```

```
theDataA [2] ["name"]
```

```
theDataA [3] ["id"]
```

```
theDataA [3] ["name"]
```

You can use this array to set the **dgData** property of a data grid.

```
set the dgData of group "DataGrid 1" to theDataA
```


Printing In LiveCode

Introduction

Printing is a vital aspect of many applications. LiveCode provides a comprehensive set of printing capabilities. Whether you want a simple print out of your stack, want to print labels or produce complex reports, LiveCode has the features you need.

LiveCode supports a number of methods of printing. You can use the *print card command* and have LiveCode manage the layout of cards on the paper. Alternatively you can use the *print into rectangle commands* which allow you to take full control over the layout of your print out. The former method is most suited to implementing simple print functionality, while the latter is better suited for more complex layout printing or printing reports. Finally, you can use the built-in field printing script library to print the contents of any text field using a simple command.

LiveCode also includes a full set of features to access and set printer device options, including options such as margins, page range and number of copies. This feature is invaluable if you want to produce a high resolution PDF file from your stack.

Controlling Printer Device Settings

LiveCode gives you full programmatic control over your printer device and any available settings.

Choosing a Printer

Use the **availablePrinters** to list the printers available on the user's system. Printers can include fax modems and networked devices. If the **availablePrinters** is empty, no printer is assigned. For example, to place a list of the available printers into a list field:

```
put the availablePrinters into field "list of printers"
```

Set the **printerName** to the printer you want to use to print. You can use any printer listed in the **availablePrinters**. This property is useful when producing an in-house utility that needs to print to a specific printer on the corporate network, or for automatically restoring the user's previous printer choice stored in a preferences file.

```
set the printerName to the cSavedPrinter of stack "My Preferences"
```

The **printerFeatures** provides a list of the features supported by the currently selected printer. Features will vary widely from device to device, but typical features may include things such as "collate", "color" and "duplex". Use this property to enable and disable output options in any custom printer settings dialog.

Choosing Output Mode (e.g. Print to File)

The **printerOutput** global property allows you to choose the output mode for subsequent printing commands. This property is set by the system to the default printer on startup and will be changed if you open a system print dialog in which the user chooses a different printer. If this property is set to *device* it will output to the physical printer. Alternatively, you can set it to a file path to print to a file. On Mac you

can set this to *preview* to create a preview.

For example, to save the current card to a file:

```
ask file "Save as:
set the printerOutput to ("file:" & it )
print this card
```

Working with Printer Dialogs

In most applications that need to print, you will want to provide a way to bring up the standard OS Print and Page Setup dialogs. Typically you would make these dialogs available from Print and Page Setup items in the File menu of your application. When the user makes changes in these dialogs the changes are made accessible to you in global properties. We discuss how to save and utilize these properties below.

Note: It is not necessary to bring up the standard OS dialogs to alter printer settings. You may set the appropriate printer setting directly by script instead.

On Linux systems you will need a recent version of GTK installed in order to display the OS printer dialog. If you don't have this installed, LiveCode will display its own printer dialog which has been built as a stack and script library. This dialog mimics the standard system printer dialog and sets the LiveCode printing global properties directly.

Tip: You may force LiveCode to use its own internal print settings dialog by setting the `systemPrintSelector` global property to false. Advanced users may customize the printer dialog that LiveCode uses by running `toplevel "print dialog"` or `toplevel "page setup"` in the Message Box. Remember to save a copy of the stack as it will be overwritten each time you upgrade your copy of LiveCode.

Important: The LiveCode print and page setup dialogs must be included in a standalone application if you use them. Ensure that the check box Print Dialog is turned on in the Standalone Application Settings dialog for your application. You do not need to include these dialogs if you only use the OS native print dialogs. For more information on including resources in your standalone application, see the chapter on Building Standalone Applications.

To bring up the standard OS printer dialog, use the `answer printer` command.

```
answer printer
```

If the user does not press the cancel button then any changes to the printer settings will be reflected in the global printing properties, discussed below.

To bring up the standard OS page setup dialog, use the `answer page setup` command.

```
answer page setup
```

Saving Printer Settings

To save or set a complete set of options relating to the current printer which includes every setting in the

OS Page Setup and Print dialogs, use the **printerSettings** global property.

The **printerSettings** property is a binary string that completely describes the current settings. The property contains the name of the printer and the settings currently in use.

You should not attempt to modify the printerSettings but rather get and set it in its entirety. To access individual printer properties, use the global printing properties described below.

When you set the **printerSettings** property to a saved value, LiveCode will choose the printer named in the property and set all of its settings to those contained within the property. If the printer cannot be found LiveCode will return the error "unknown printer" in the result. If the printer is found but the settings are not valid then LiveCode will choose the printer and reset it to default values.

Note: You must save a separate copy of the printerSettings property for each printer or OS you intend to use. The printerSettings property cannot be transferred between platforms. For example, a printerSettings property generated on a Windows computer cannot be used on Mac – even for the same printer. If you want to alter settings across different platforms and printer types use the global printing properties described below. Use the printerSettings for convenience when you know you will be using the same printer and want to save all the settings, or where you are setting esoteric properties not listed in the global printing properties described below.

To save the current printer settings into a custom property stored on the current stack:

```
set the cSavedPrinterSettings of this stack to the printerSettings
save this stack
```

Then to restore these settings:

```
set the printerSettings to the cSavedPrinterSettings of this stack
```

Paper Related Options

This section discusses how you get and set paper related options – the rectangle area of the paper, the paper size, the orientation and the scale to use to print onto the paper. These paper options apply to all types of printing, i.e. card, field and layout printing.

Use the **printRectangle** to get the printable rectangle area within the paper (returned in device co-ordinates). This property takes into account any settings applied by the user in the Page Setup and Printer dialogs including the print orientation (e.g. landscape or portrait). The rectangle is represented left,top,right,bottom and is always relative to the top left of the page – thus the top left will always be 0,0. The printRectangle will always be within the **printPaperRectangle** – the rectangular area of the sheet of paper.

The printRectangle property is read only and cannot be set directly – to alter it you must set other options relating to the paper, for example the **printPaperOrientation**(discussed below).

Do not confuse the **printMargins** and other card layout printing properties with paper properties such as the **printRectangle**. The **printMargins** only applies to printing cards using LiveCode's automatic card layout capabilities (discussed below). Thus the **printMargins** has no effect on **printRectangle**.

Use the **printPaperOrientation** to get and set the orientation of your print out. This property may be set to one of the following values:

portrait: rotated 0 degrees.

landscape: rotated 90 degrees clockwise.

reverse portrait: rotated 180 degrees clockwise.

reverse landscape: 270 degrees clockwise.

```
set the printPaperOrientation to "landscape"
```

Use the **printPaperScale** property to apply a scale factor to your print out after all other settings have been taking into account.

The **printPaperScale** is applied after all other layout and scaling options. For example, if you have used the layout printing features to print a series of cards at 50% scale, then set the **printPaperScale**, this factor will be applied to the entire layout after the card layout scaling has been calculated.

To print a range between 1 and 100% set the **printPaperScale** to a number between 0 and 1. To print at 200% set the **printPaperScale** to 2.

```
set the printPaperScale to 0.5 -- 50%
```

Job Related Options

This section discusses how to get and set job related options – the number of copies, duplex printing, collation, color, title and printable area.

Important: The available job options all depend on what the currently selected printer supports (use the **printerFeatures** property, described above, to retrieve a list of features supported by the current printer).

Use the **printCopies** property to get and set the number of copies to print. The **printCopies** should be set to a value of 1 or more.

```
set the printCopies to 5 -- print 5 copies
```

Use the **printDuplex** property to tell the printer to print double sided. This property may be set to any of the following values:

- *none*: no double-sided printing
- *short edge*: double-sided printing with tumble (flip the non-facing page)
- *long edge*: double-sided printing without tumble.

set the **printDuplex** to "short edge"

Use the **printCollate** property to specify whether to interleave multiple copies of a print job. If a print job has three pages, P1, P2 and P3, with **printCollate** set to *true* and **printCopies** set to 2 the output order will be P1, P2, P3, P1, P2, P3. With **printCollate** set to *false* the output will be P1, P1, P2, P2, P3, P3.

```
set the printCollate to true
```

Use the **printColors** property to specify whether to print in color or not. If "color" is not among the lines of the **printerFeatures** then this property will have no effect and all print jobs will be printed in monochrome. This property may be set to either *true* or *false*.

For example, to check if color printing is supported on the current printer and use it if it is:

```
if "color" is among the lines of the printerFeatures then
    set the printColors to true
end if
```

Use the **printTitle** property to specify the name of the next print job in the system printer queue. Setting this property to match the name of the user's document will ensure that the user is able to recognize it in the system printer queue utility. If the **printTitle** is empty at the start of a printing loop, the title of the defaultStack will be used.

```
set the printTitle to "My Report 1"
```

Use the **printRectangle** property to determine the printable region of the physical page as returned by the printer. This rectangle will always be contained within the **printPaperRectangle**. Thus you should use the **printRectangle** and not the **printPaperRectangle** when calculating a print layout. The **printPaperRectangle** is useful if you are generating a print preview and want to show the entire area of the paper including any margin areas that cannot be printed on. This property is read only and cannot be set directly.

Printer Font Metrics (Windows)

Windows systems sometimes use different versions of the same font for displaying text on screen and printing. This can result in layouts and line breaks differing between the screen display and the printed output. To prevent this from happening, you can tell LiveCode to use the printer fonts for display on screen. To do this, set a stack's **formatForPrinting** property to true.

Do:

- Set the **formatForPrinting** stack property to true before loading a stack in memory. If the stack is already loaded, set this property to true then save and reload it. (Save then use *Close and Remove from Memory* in the File menu).
- Create a stack off screen (with **formatForPrinting** set to true) with your print layout template and copy text into it prior to printing.
- Set the **formatForPrinting** before doing any print layout related calculations on the stack.
- Set the **formatForPrinting** to true on any print preview stack being displayed to the user.

Don't:

- Allow the user to directly edit text in fields whose **formatForPrinting** is set to true. Attempting to do this may cause display anomalies. Set this property to false and reload the stack first.
- Generally use stacks with **formatForPrinting** set to true for display on screen, as this will show text that has been optimized for print display (instead of screen display), which is harder to read on screen.
- Use this property on other platforms – Windows is the only platform that uses different fonts on

screen vs. in a print out.

- Use the **windowBoundingRect** property to constrain display of a stack who's **formatForPrinting** has been set to true – this property will be ignored when the stack is opened or maximized.

Printing a Card

Once you have set your printer, paper and job options (above) you are now ready to use one of the print commands to start printing. At its simplest, the **print** card command allows you to print a card. Later we will discuss ways of printing more complex layouts, fields and text.

```
print this card -- prints the current card
print card 12 -- prints card 12
```

For more details on how to specify which cards to print, see the **print** command in the *LiveCode Dictionary*.

To print a scale between 1 and 100% set the **printScale** to a number between 0 and 1. To print at 200% set the **printScale** to 2.

The **printScale** applies to each card that you print. It is not related to the **printPaperScale** which is applied to the entire print job after all other scaling calculations have been applied. Thus you may set the **printPaperScale** to 0.5 to print at 50%, then print individual cards at different **printScale** values. With a **printPaperScale** of 0.5, a **printScale** of 2 would result in that card being printed at 100%.

When printing a card, use the **printMargins** to specify the margins around the border of the card on the page.

Note: When calculating placement on the printed page, all calculations assume that there are 72 dots per inch – regardless of platform or printer device. LiveCode will automatically adjust the print out for resolution of the actual device. This makes it simple to calculate your printed layout.

```
-- a one-inch margin on each side
set the printMargins is set to 72,72,72,72
```

The **printMargins** only applies when using **print card** directly. It does not have any effect on printing cards into a layout (described below).

The **printCardBorders** property specifies whether or not the bevel border around the edge of a card should appear in the print out.

Card Layout Options

When using the basic **print card** form of the **print** command, there are two layout options that allow you to customize the positioning of cards on the printed page. If you require further flexibility, see the section on printing a layout, below.

Use the **printRowsFirst** property to specify whether cards should be printed across and down or down then across.

Consider a simple example of printing multiple cards – this example is useful for printing labels. In this example we have a stack that contains 8 cards, each one containing a mailing label. If you want to try out

this example:

- Create a stack and size it to be small – the size of a mailing label
- Create a single field, and in the field Inspector turn off the *Shared Text* property
- Group the field and in the group property *Inspector* turn on *Behave as Background*
- Turn on *Select Grouped* on the Toolbar and select the field
- Place the contents of the first mailing label into the *Contents* tab of the *Inspector*
- Create 8 more cards, and in each select the field and place the contents of a different label

Thus we have a stack that looks like the figure below.



Now we will implement the printing commands. If this was a real application you would probably want to put these in a Print command in the File menu. In this instance you may execute the following in the multi-line message box (open the Message Box then press the second icon to get the multi-line pane).

```
-- allow the user to choose printer output options
answer printer
print 9 cards
```

Press enter to execute the commands.

The resulting print out will look like the figure below.



If we modify the print commands to include an additional line to turn off the `printRowsFirst`:

```
answer printer
set the printRowsFirst to false
print 9 cards
```

The resulting print out will look like the figure below.



Use the **printGutters** property to specify the margin between each card. By default the `printGutters` are set to 36,36 or one half inch horizontally and vertically.

In the following example, we print the same label stack but reduce the space between each label to 1/10th of an inch. To make it easy to see the difference we also turn on printing of card borders using the `printCardBorders` property.

```
answer printer
set the printGutters to 7,7
set the printCardBorders to true
print 9 cards
```

The resulting print out will look like the figure below.

Printing Fields & Text

To print a field, use the **revPrintField** command. This command takes a single parameter, a reference to a field. This command only allows printing a single field. If you need to include a header and footer or text you have constructed programmatically, see the **revPrintText** command below.

```
revPrintField the long id of field "text document"
```

revPrintField is implemented as a script library located in the LiveCode IDE. The script library creates an invisible stack, sets the rectangle of that stack to the current paper size, sets the **formatForPrinting** to true, creates a field, then copies the contents of the field you specify into this invisible stack. It then prints the field one page at a time, scrolling the text after each page. Advanced users can locate this library script by going to the *Back Scripts* tab in the Message Box, turning on the checkbox for *Show LiveCode UI Back Scripts*, then editing the script of stack "revPrintLibrary". The *revPrintField* handler is near the top of the script.

Use the **revShowPrintDialog** command to control whether the system printer and page setups dialogs should be shown by **revPrintField** or **revPrintText**.

```
-- show the system printer dialog, but not page setup
revShowPrintDialog true, false
revPrintField the long id of field "text document"
```

Use the **revPrintText** command to print plain or styled text together with an optional header and footer.

```
revPrintText textToPrint [,headerText [,footerText [,fieldTemplate]]]
```

The *textToPrint* is anything which evaluates to a string. If you want to printed styled text, pass HTML instead of plain text. (To convert a field containing styled text to a HTML use the **htmlText** property.)

The *headerText* and *footerText* contains the text to use as a header and footer. You may include an expression that is computed for each page. For more details on using expressions, see the *LiveCode Dictionary* entry for **revPrintText**.

The *fieldTemplate* parameter allows you to specify a field reference to use. Fonts in the print out will be inherited from this field.

Printing a Layout

If you need to print a more complex layout than allowed with the basic print card command or text printing commands (described above), you can use the **print card into rect** syntax to create any sort of layout you choose.


```
print card from topLeft to rightBottom into pageRect
```

The *topLeft* is the top left coordinate of the current card to start printing at.

The *rightBottom* is the bottom right coordinate of the current card to stop printing at.

The *pageRect* is the rectangular area on the paper to print into.

`printMargins` only applies when using `print card` directly. It does not have any effect on printing cards into a layout. Use the **printRectangle** to get the printable area when working with layout printing.

For example, let's say that we want to print the text field from the middle of the stack in the figure below. (You can load the stack shown in the picture by going to your LiveCode installation folder then opening Resources-> Examples-> SQLite Sampler.rev.) We want the output to scale to take up the entire width of the paper and half the height.

```
local tRect
put 0,0,item 3 of the printRectangle, \
    round(item 4 of the printRectangle / 2) into tRect
print this card from the topLeft of field "theText" \
    to the bottomRight of field "theText" \
    into tRect
```

This results in the print out shown in the figure below.

You can construct a complex layout taking components from multiple stacks by printing a sequence of rectangles onto the same page. For example, you may have a stack that contains a standard header and footer, another that contains a logo, and a layout that contains text. Use the **open printing** command to start a print job, then print each element into the appropriate rectangle on the paper. Then use the **close printing** command to send the print job to the printer. The example in the figure below shows two stacks with printable regions that we want to combine onto a single sheet of paper.

To print these onto a single sheet:

```

answer printer -- show the system print settings dialog
open printing -- start a print job
-- work with the header stack
set the defaultStack to "header"
-- print the header field onto the top left of the paper
print this card from the topLeft of field "header" \
    to the bottomRight of field "header" \
    into the rect of field "header"
-- save the bottom of the header
put the bottom of field "header" into tHeaderBottom
set the defaultStack to "report editor"
-- print the table field below the header
print this card from the `topLeft` of field "report table" \
    to the bottomRight of field "report table" \
    into 0,tHeaderBottom,the`right of field "report table", \
    the bottom of field "report table" + tHeaderBottom
-- send `the`job to `the`printer
close`printing

```

Printing a complex layout

To print a more complicated layout, create a stack and set its rectangle to the current **printRectangle**. Add rectangular areas for each component you will be printing. Then set *Geometry* properties (see the section on the *Geometry Manager*, in the **LiveCode Script** guide for more information) on each of these rectangles so they resize correctly when the stack is scaled. Set up your print routine so that you open this stack invisibly then resize it to the **printRectangle**. This will trigger the geometry routines and scale the rectangular areas correctly. Then run your sequence of print commands to print into each rectangle.

In the figure below, we have set the size of the stack to the **printRectangle** then added 4 rectangle graphics. We have named each graphic and turned on the *Show Name* property for each so you can see the name.

Next, we set *Geometry* properties for each of the rectangle graphics. The header graphic is set to scale relative to the right and bottom, with a minimum size limit of 100 pixels (see the figure below). The body graphic is set to link the top edge to the header graphic, the right and bottom to the stack (see the figure below). The footer 1 graphic is set to scale the right edge and position the bottom edge. And footer 2 is set to position both the right and bottom.

To make the printing template stack take on the size of the paper, we can add the following handler to the stack script:

```

on preOpenStack
    set the width of this stack to (item 3 of the printRectangle - item 1 of the
printRectangle)
    set the height of this stack to (item 4 of the printRectangle - item 2 of the
printRectangle)
end preOpenStack

```

We now have a working print template stack. All that remains is to write the script that prints into the rectangles:

```

-- prepare to load this stack off screen
hide stack "print layout"

-- this will trigger the stack to resize, which run the geometry
-- routines, giving us the correct values for each rectangle
go stack "print layout"

-- now we store these rectangle coordinates in variables
put the rect of graphic "header" into tHeaderRect
put the rect of graphic "body" into tBodyRect
put the rect of graphic "footer 1" into tFooter1Rect
put the rect of graphic "footer 2" into tFooter2Rect

-- we can close the layout stack as its no longer needed
close stack "print layout"

-- load the system printer dialog to allow the user
-- to choose number of copies, etc.
answer printer

-- start the print job
open printing

-- set the stack we are working on to a stack containing the header
-- you could use any stack or region within a card here
set the defaultStack to stack "header graphics"

-- print from the rectangle of our header group
-- into the rectangle we stored earlier
-- we could use a graphic or any rectangular area instead of a group
print this card from the topLeft of group "header" \
    to the bottomRight of group "header" into tHeaderRect

set the defaultStack to "body contents"

print this card from the topLeft of group "body" \
    to the bottomRight of group "body" into tBodyRect

set the defaultStack to "footer1"

print this card from the topLeft of group "footer1" \
    to the bottomRight of group "footer1" into tFooter1Rect

```

```

to the bottomRight of group footer1 into tFooterRect

set the defaultStack to "footer2"

print this card from the topLeft of group "footer2" \
  to the bottomRight of group "footer2"

-- send the print job to the printer
close printing

-- check to see if the user cancelled or there was an error
if the result is "cancel" then
  -- insert any code you need here to handle cancellation
else if the result is not empty then
  -- bring up an error dialog
  answer "Printer Error"
else
  -- insert any code you need here to handle success
end if

```

For more information on how to print multiple pages of a complex layout, see the section on *Printing Multiple Pages*, below. For information on printing scrolling text fields into an area within a layout, see the section on *Working with Scrolling Fields when Layout Printing*, below.

Printing Multiple Pages

Multiple pages using card printing

To print multiple pages when printing cards, simply specify which cards you want to print as part of the print command.

```
print {range}
```

Examples:

```

-- prints the current card
print this card
-- prints all cards in the current stack
print all cards
-- prints the next 10 cards, starting with the current card
print 10 cards
-- prints card 3 to 7 of the current stack
print card 3 to 7 print marked cards
-- prints all cards where the mark property is true
print marked cards

```

Multiple pages when using layout printing

To print multiple pages when printing layouts, use the **open printing** command to open a print job. Then

print the layout for the first page (see above). Then use the **print break** command to insert a page break into the print job. Then lay out the second page and so on. Finally, use the **close printing** command to send the print job to the printer.

Working with Scrolling Fields when Layout Printing

To print a single scrolling text field, use the **revPrintText** command (see above for more information). If you want to incorporate the contents of a scrolling field within a layout, use the **pageHeights** property to scroll the field each time you print a page, then **print break** to move to the next page.

The **pageHeights** returns a list of values to indicate how far a scrolling field needs to be scrolled to avoid clipping a line of text on each page of your print out. (You should use this feature in conjunction with the **formatForPrinting** property, above.)

```
-- store a list of pageHeights
put the pageHeights of field "body text" into tHeightsList
-- scroll the field to the start
set the scroll of field "body text" to 0
-- start the print job
open printing
repeat for each line l in tHeightsList
-- clip the field to the bottom of the last visible line
set the height of field "body text" to l
-- print the field rectangle
-- you may want to add an output "into" rectangle

print this card from the topLeft of field "body text" \
  to the bottomRight of field "body text"

-- print a new page
print break
end repeat
-- send the job to the printer
close printing
```

Important: Set the Lock Location (**lockLoc**) property of the field to true before setting the height in the loop above to avoid the field “drifting” each time you alter the height.

Tip: Turn off the scroll bar properties (**hScrollbar** and **vScrollbar**) of the field before printing and set the border width to 0 if you want to avoid printing a border or scroll bar.

You can incorporate scrolling fields into a template print layout stack (see the section *Printing a Complex Layout* above) to make it easier to manage printing a complex layout. Create a field in your template print stack instead of a graphic, turn off the scroll bar, set the border width to 0, the lock location to true and then the geometry properties as per the section above. At the start of each print job, copy the text font and size by using the **textFont** and **textSize** properties, then the contents of the text field you want to print using the **htmlText** property.

Working with Print Ranges

Use the **printRanges** property to get a list of pages that the user has selected in the printer settings dialog. Use this property when printing to avoid printing pages the user has not selected. To use this property, open a system printer dialog, then store the **printRanges** in a variable. Then set the **printRanges** to "all", then send only the pages that were selected (stored in the variable) to the printer.

Note: If you ignore the `printRanges` property LiveCode will handle this setting automatically. Simply send every page to the printer as normal and LiveCode will ignore the pages the user has not selected in the print dialog. Handle this option manually only if you are printing an extremely complex layout and want to save processing time building the layout for every unselected page.

Use the **`printPageNumber`** to get the number of the page currently being printed during your printing loop.

Printing a Browser Object

To print the contents of a browser object, use the **`revBrowserPrint`** command. For more information, see the **`revBrowserPrint`** command in the *LiveCode Dictionary*.

Processing Text and Data

Introduction

LiveCode has first-class text and data processing capabilities. LiveCode's unique chunk expressions – the ability to refer to text using English-like statements like "word 3 to 5 of myVariable", combined with other powerful features which include regular expressions, XML processing, associative arrays, data encoding and decoding functions and compression and encryption algorithms – make it easy and simple to process text and data of any complexity. This chapter is a reference guide - go [here](#) for more resources.

The section *Processing Text and Data* in the *Sample Scripts* within the product documentation contains additional code examples.

Using Chunk Expressions

Chunk expressions are the primary method of working with text in LiveCode. A chunk is an English-like way of describing an exact portion of text. You can use chunks both to retrieve a portion of text, and to edit text. This topic defines the types of chunks you can address and describes the syntax for specifying them.

Types of Chunks

The common types of chunks are the **character**, **word**, **line**, or **item**. An item can be delimited by any character you specify. In addition, the **token** chunk is useful when parsing script data.

Here is an example of a chunk expression using the **word** chunk:

```
put word 1 to 3 of field "text" into myVariable
```

Using Chunks with Containers

You can use a chunk of a *container* anywhere you use an entire container. For example, you can use the **add** command to add a number to a line of a field:

```
add 1 to word 3 of field "Numbers"
```

You can also use chunk expressions to replace (using the **put** command) or remove (using the **delete** command) any portion of a container.

Using Chunks with Properties

You *can* use chunk expressions to *read* portions of a *property* (such as the **script** property). However, since you change a property with the **set** command rather than the **put** command, you *can't* use a chunk expression to *change* a part of a property's value. Instead, **put** the property value into a variable, use the chunk expression to change the variable, then **set** the property to the variable's contents. The following example shows how to change the third line of an object's **script** property:

```

put the script of me into tempScript
put "-- Last changed by Jane" into line 3 of tempScript
set the script of me to tempScript

```

The basic chunk types

There are four chunk types that are probably the most useful for basic text processing: **character**, **segment**, **item** and **line**.

The Character Chunk

A **character** is a single character, which may be a letter, digit, punctuation mark, or control character.

You can use the abbreviation **char** as a synonym for **character** in a chunk expression.

Important: The character chunk corresponds to the notion of grapheme in the Unicode standard.

The Segment Chunk

A **segment** is a string of characters delimited by **space**, **tab**, or **return** characters or enclosed by double quotes.

Note: **Word** is a synonym of segment.

The Item Chunk and the itemDelimiter Property

By default, an **item** is a string of characters delimited by commas.

Items are delimited by a string specified in the **itemDelimiter** property. You can change the default *comma* to create your own chunk type by setting the **itemDelimiter** property to any string.

The Line Chunk and the lineDelimiter Property

By default, a **line** is a string of characters delimited by the **return** character.

Lines are delimited by string in the **lineDelimiter** property. By default, the **lineDelimiter** is set to **return**, but you can create your own chunk type by setting the **lineDelimiter** property to any string.

Other chunks

There are also some chunks which are useful for more advanced text processing. These are:

- paragraph
- sentence
- trueWord
- codepoint
- codeunit
- byte

A **token** is a string of characters delimited by certain punctuation marks. The **token** chunk is useful in parsing LiveCode statements, and is generally used only for analyzing scripts.

The **sentence** and **trueWord** chunk expressions facilitate the processing of text, taking into account the different character sets and conventions used by various languages. They use the ICU library, which

uses a large database of rules for its boundary analysis, to determine sentence and word breaks.

The **paragraph** chunk is currently identical to the existing **line** chunk, however in the future **paragraph** chunks will also be delimited by the Unicode paragraph separator.

The **codepoint** chunk type allows access to the sequence of Unicode codepoints which make up the string. The **codeunit** chunk type allows direct access to the UTF-16 code-units which notionally make up the internal storage of strings. The **codeunit** and **codepoint** chunk are the same if a string only contains unicode codepoints from the Basic Multilingual Plane.

The **byte** chunk is an 8-bit unit and should be used when processing binary data.

For more information on the above chunk types, please consult the LiveCode Dictionary.

Specifying a Chunk

The simplest chunk expression specifies a single chunk of any type. The following statements all include valid chunk expressions:

```
get char 2 of "ABC" *-- yields "B"*
get segment 4 of "This is a test" *-- yields "test"*
get line 7 of myTestData
put "A" into char 2 of myVariable
```

You can also use the ordinal numbers **first**, **last**, **middle**, **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**, **eighth**, **ninth**, and **tenth** to designate single chunks. The special ordinal **any** specifies a *random* chunk.

```
put "7" into last char of "1085" -- yields "1087"
```

Negative Indexes in Chunk Expressions

To count *backwards* from the end of the value instead of forward from the beginning, specify a *negative* number. For example, the number -1 specifies the last chunk of the specified type, -2 specifies the next-to-last chunk, and so forth. The following statements all include valid chunk expressions:

```
get item -1 of "feather, ball, cap" -- yields "cap"
get char -3 of "ABCD" -- yields "B"
```

Complex Chunk Expressions

More complex chunk expressions can be constructed by specifying a chunk within another chunk. For example, the chunk expression `segment 4 of line 250` specifies the fourth segment of line 250.

When combining chunks of different types to construct a complex chunk expression, you must specify the chunk types in order. The following statements all include valid chunk expressions:

```
char 7 of segment 3 of myValue
segment 9 of item 2 of myValue
last char of segment 8 of line 4 of myValue
```

These, however, are not valid chunk expressions:

```
segment 8 of char 2 of myValue --chars can't contain segments
item 9 of first segment of myValue --segments can't contain items
line 3 of last item of myValue --items can't contain lines
```

The full hierarchy is as follows:

```
paragraph > sentence > line > item > segment >
  trueWord > token > character > codepoint > codeunit > byte
```

Using Parentheses in Chunk Expressions

You use parentheses in chunk expressions for the same reasons they're used in arithmetic:

To make a complex expression clearer.

To change the order in which the parts of the expression are evaluated.

For example, consider the following statement:

```
put item 2 of segment 3 of "a,b,c i,j,k x,y,z" -- BAD
```

The desired result is "y", the second item in the third segment. But the statement above causes an execution error, because it asks for an item of a segment, and segments can't contain items. You can obtain the desired result by using parentheses to change the order of evaluation:

```
put item 2 of (segment 3 of "a,b,c i,j,k x,y,z") -- good
```

In the example above, LiveCode gets the third segment first, then gets the second item in that segment. By adding parentheses around (segment 3 of "a,b,c i,j,k x,y,z"), you force LiveCode to evaluate that part of the chunk expression first. The value of the expression in parentheses is "x,y,z", and item 2 of "x,y,z" is "y".

As with arithmetic expressions, the parts of a chunk expression that are in parentheses are evaluated first. If parentheses are nested, the chunk expression within the innermost set of parentheses is evaluated first. The part that is enclosed in parentheses must be a valid chunk expression, as well as being part of a larger chunk expression:

```
put line 2 of segment 1 to 15 of myValue -- won't work
put line 2 of segment (1 to 15 of myValue) -- won't work
put line 2 of segment 1 to 15 (of myValue) -- won't work
put line 2 of (segment 1 to 15 of myValue) -- works!
```

The first of the above examples doesn't work for much the same reason as the previous example: segments can't contain lines. The second and third examples don't work because neither "1 to 15 of myValue" nor "of myValue" is a valid chunk expression. However, "segment 1 to 15 of myValue" is a valid chunk expression, so the last example works.

Nonexistent Chunks

If you request a chunk number that doesn't exist, the chunk expression evaluates to empty. For example, the expression `char 7 of "AB"` yields empty.

If you attempt to change a chunk that doesn't exist, what happens depends on what kind of chunk you specify:

Nonexistent character or segment:

Putting text into a character or segment that doesn't exist *appends* the text to the end of the container, without inserting any extra spaces.

Nonexistent item:

Putting text into an item that doesn't exist *adds* enough **itemDelimiter** characters to bring the specified item into existence.

Nonexistent line:

Putting text into a line that doesn't exist *adds* enough **return** characters to bring the specified line number into existence.

Specifying a Range

To specify a portion larger than a single chunk, you specify the beginning and end of the range. These are all valid chunk expressions:

```
get char 1 to 3 of "ABCD" -- yields "ABC"
get segment 2 to -1 of myValue -- second segment to last segment
put it into line 7 to 21 of myValue -- replaces
```

The start and end of the range must be specified as the same chunk type, and the beginning of the range must occur *earlier* in the value than the end. The following are not valid chunk expressions:

```
char 3 to 1 of myValue -- won't work
-- end cannot be greater than start

char -1 to -4 of myValue -- won't work
-- 4th from last comes before last
```

Important: When using negative numbers in a range, remember that numerically, -x comes after -x+1. For example, -1 is greater than -2, and -4 is greater than -7. The greater number must come **last** in order to create a valid range.

Counting the Number of segments, Lines or Items

The **number** function returns the number of chunks of a given type in a value. For example, to find out how many lines are in a variable, use an expression such as:

```
the number of lines in myVariable
```

You can also nest chunk expressions to find the number of chunks in a single chunk of a larger chunk type:

```
the number of chars of item 10 of myVariable
```

Comparing and Searching

LiveCode provides a number of ways of comparing and searching text. For most types of searching and comparing, you will find chunk expressions easy and convenient. However, if you have complex searching needs, you may prefer to use Regular Expressions, covered in the next section.

Checking if a Part is within a Whole

You use the **is in** operator to check if some text or data is within another piece of text or data. You can use the reverse **is not in** operator to check if text or data is not within another piece of text or data.

```
"A" is in "ABC" -- evaluates to true
"123" is in "13" -- evaluates to false
```

You can also use the **is in** operator to check whether some text or data is within a specified chunk of another container.

```
"A" is in item 1 of "A,B,C" -- evaluates to true
"123" is in segment 2 of "123 456 789" -- evaluates to false
```

Case Sensitivity

Comparisons in LiveCode are case insensitive by default (except for Regular Expressions, which have their own syntax for specifying whether or not a match should be case sensitive). To make a comparison case sensitive, set the **caseSensitive** property to true. For more details, see the *caseSensitive* property in the *LiveCode Dictionary*.

Checking if text is True, False, a Number, an Integer, a Point,

a Rectangle, a Date or a Color

Use the **is a** operator for checking whether the user has entered data correctly and for validating parameters before sending them to a handler. The **is an** operator is equivalent to the **is a** operator.

A value **is a**:

- **boolean** if it is either true or false
- **integer** if it consists of digits (with an optional leading minus sign)
- **number** if it consists of digits, optional leading minus sign, optional decimal point, and optional "E" or "e" (scientific notation)
- **point** if it consists of two numbers separated by a comma
- **rect** if it consists of four numbers separated by commas
- **date** if it is in one of the formats produced by the date or time functions
- **color** if it is a valid color reference

The text you are checking can contain leading or trailing white space characters in all the types except boolean. For example:

```
" true" is true -- evaluates to false
```

The **is a** operator is the logical inverse of the **is not a** operator. When one is true, the other is false.

```
"1/16/98" is a date -- evaluates to true
1 is a boolean -- evaluates to false
45.4 is an integer -- evaluates to false
"red" is a color -- evaluates to true
```

Tip: To restrict a user to typing numbers in a field, use the following script

```
on keyDown pKey
  if pKey is a number then pass keyDown
end keyDown
```

The **keyDown** message will only be passed if the key the user pressed is a number. If you trap a **keyDown** message and don't pass it, the key will not be entered into the field. For more details, see the **keyDown** message in the *LiveCode Dictionary*.

Check if a segment, Item or Line Matches Exactly

The **is among** operator tells you whether a whole chunk exists exactly within in a larger container. For example, to find out whether the whole segment "free" is contained within a larger string, use the **is among** operator:

```
"free" is among the segments of "Live free or die" -- true
"free" is among the segments of "Unfree world" -- false
```

The second example evaluates to false because, although the string "free" is found in the value, it's a portion of a larger segment, not an entire segment.

Check if one String Starts or Ends With Another

To check if one string begins with or ends with another, use the **begins with** or **ends with** binary operators. For example:

```
"foobar" begins with "foo" -- true
"foobar" ends with "bar" -- true
line 5 of tList begins with "the"
```

Replacing Text

To replace one string with another, use the **replace** command. (If you want the search string to contain a regular expression, see the section on the *replaceText* command below instead.)

```
replace "A" with "N" in thisVariable -- changes A to N
```

To delete text using replace, replace a string with the empty constant.

```
replace return with empty in field 1 -- runs lines together
```

To preserve styling when replacing one string with another in a field, use the **preserving styles** clause. replace "foo" with "bar" in field "myStyledField" preserving styles -- replaces instances of "foo" with "bar" in the given field, -- "bar" will retain the styling of instances of "foo"

For more details, see the *replace* command and the *replace in field* command in the *LiveCode Dictionary*.

Retrieving the Position of a Matching Chunk

The **offset**, **itemOffset**, **tokenOffset**, **trueWordOffset**, **segmentOffset**, **lineOffset**, **sentenceOffset** and **paragraphOffset** functions can be used to locate the position chunks within a larger container. For example, this expression returns the character number where the letter "C" was found:

```
get offset("C", "ABC") -- returns 3
```

To check if an item, line or word matches *exactly* using offset, set the **wholeMatches** property to true.

Chunks Summary

A chunk expression describes the location of a piece of text in a longer string.

Chunk expressions can describe **characters**, **items**, **tokens**, **trueWords**, **segments**, **lines**, **sentences** and **paragraphs** of text.

To count backward from the end of a string, use negative numbers. For example,

```
segment -2 -- indicates the second-to-last segment
```

You can combine chunk expressions to specify one chunk that is contained in another chunk, as in

```
segment 2 of line 3 of myVariable
```

For a range of chunks, specify the start and end points of the range, as in line 2 to 5 of myVariable

To check if a chunk is within another, use the **is in** operator. To check if a chunk **is a** specified type of data, use the **is a** operator. To check if a chunk starts or ends with another uses the **begins with** or **ends with** operators.

To check if a chunk is contained exactly within a string use the **is among** operator. To get an index specifying where a chunk can be found in a container, use the **\Offset** functions described above. To match only a complete chunk within a string, set the `wholeMatches` to true before using the offset functions.

Regular Expressions

Regular expressions allow you to check if a *pattern* is contained within a string. Use regular expressions when one of the search or comparison chunk expressions does not do what you need (see the section on *Comparing and Searching* above).

LiveCode supports searching for a pattern, replacing a pattern, or filtering the lines in a container depending on whether or not each line contains the pattern. Regular expressions use PERL compatible or "PCRE" syntax. Figure 52, below, shows the supported syntax. For more details on the supported syntax, see the [PCRE manual](#)

Searching using a Regular Expression

Use the **matchText** function to check whether a string contains a specified pattern.

```
matchText(string, regularExpression[, foundTextVarsList])
```

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

The *regularExpression* is any expression that evaluates to a regular expression.

The optional *foundTextVarsList* consists of one or more names of existing variables, separated by commas.

```
matchText("Goodbye", "bye") -- returns true
matchText("Goodbye", "^Good") -- also returns true
matchText(phoneNumber, "([0-9]+)-([0-9]+-[0-9]+)", areaCode, phone)
```

For more details on this function see the *matchText* function in the *LiveCode Dictionary*.

If you need to retrieve the positions of the matched substrings in the optional *foundTextVarsList*, use the *matchChunk* function instead of the *matchText* function. These functions are otherwise identical.

Replacing using a Regular Expression

Use the **replaceText** function to search for a regular expression and replace the portions that match. If you simply want to replace text without using a regular expression, see the **replace** command instead.

```
replaceText(stringToChange,matchExpression,replacementString)
```

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

The *matchExpression* is a regular expression.

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

```
replaceText("malformed","mal","well")--returns "wellformed"
replaceText(field "Stats",return,comma)-- makes comma-delimited
```

For more details, see the *replaceText* function in the *LiveCode Dictionary*.

Regex	Rule	Example
[chars]	matches any one of the characters inside the brackets	A[BCD]E matches "ACE", but not "AFE" or "AB"
[^chars]	matches any single character that is not inside the brackets	[^ABC]D matches "FD" or "ZD", but not "AD" or "CD"
[char-char]	matches the range from the first char to the second char. The first char's ASCII value must be less than the second char's ASCII value	A[B-D] matches "AB" or "AC", but not "AG"
.	matches any single character (except a linefeed)	[A-Z0-9] matches any alphanumeric character A.C matches "ABC" or "AGC", but not "AC" or "ABDC"
^	matches the following character at the beginning of the string	^A matches "ABC" but not "CAB"
\$	matches the preceding character at the end of a string	B\$ matches "CAB" but not "BBC"

Regex	Rule	Example
*	matches zero or more occurrences of the preceding character or pattern	<p>ZA*B matches "ZB" or "ZAB" or "ZAAB", but not "ZXA" or "AB"</p> <p>[A-D]*G matches "AG" or "G" or "CAG", but not "AB"</p> <p>ZA+B matches "ZAB" or "ZAAB", but not "ZB"</p>
+	matches one or more occurrences of the preceding character or pattern	<p>[A-D]+G matches "AG" or "CAG", but not "G" or "AB"</p> <p>ZA?B matches "ZB" or "ZAB", but not "ZAAB"</p>
?	matches zero or one occurrences of the preceding character or pattern	<p>[A-D]?G matches "AG" or "CAGZ", but not "G" or "AB"</p>
**\	**	<p>matches either the pattern before or the pattern after the \</p> <p>B matches "A" or "B"</p> <p>[XYZ] matches "AY" or "CX", but not "AA" or "ZB".</p>
\	Causes the following character to be matched literally, even if it has special meaning in a regular expression	<p>A\.C matches "A.C", but not "A\C" or "ABC"</p> <p>matches "\"</p>
any other character	matches itself	<p>ABC matches "ABC"</p>

Figure 50 – Regular Expression Syntax

Filtering using a Wildcard Expression

Use the **filter** command to remove lines in a container that either do, or do not match a specified wildcard expression. Wildcard expressions are similar to regular expressions.

```
filter container {with | without} wildcardExpression
```

The *container* is any expression that evaluates to a container.

The *wildcardExpression* is a pattern used to match certain lines.

```
filter myVariable with "A?2"
filter me without "\*[a-zA-Z]\*"
```

For more details, including the format of wildcard expressions, see the *filter command* in the *LiveCode Dictionary*.

International Text Support

All LiveCode's text processing capabilities extend seamlessly to international text. This includes the ability to render and edit Unicode text and convert between various encoding types.

What are Text Encodings?

Fundamentally computers use numbers to store information, converting those numbers to text to be displayed on the screen. A text encoding describes which number converts to a given character. There are many different encoding systems for different languages. Below is a table containing examples of some common encodings.

Encoding	Representation	Description
ASCII	Single byte – English	ASCII is a 7-bit encoding, using one byte per character. It includes the full Roman alphabet, Arabic numerals, Western punctuation and control characters. See http://en.wikipedia.org/wiki/ASCII for more information.
ISO8859	Single byte	ISO8859 is a collection of 10 encodings. They are all 8-bit, using one byte per character. Each shares the first 128 ASCII characters. The upper 80 characters change depending on the language to be displayed. For example ISO8859-1 is used in Western Europe, whereas ISO8859-5 is used for Cyrillic. NB: LiveCode only supports ISO8859-1. You should use Unicode to represent other languages, converting if necessary (see below).
Windows-1252	Single byte – English	This is a superset of ISO8859-1 which uses the remaining 48 characters not used in the ISO character set to display characters on Windows systems. For example curly quotes are contained within this range.
MacRoman	Single byte – English	MacRoman is a superset of ASCII. The first 128 characters are the same. The upper 128 characters are entirely rearranged and bear no relation to either Windows-1252 or ISO8859-1. However while many of the symbols are in different positions many are equivalent so it is possible to convert between the two.
UTF-16	Double byte – Any	UTF-16 typically uses two bytes per code point (character) to display text in all the world's languages (see <i>Introduction to Unicode</i> , below). UTF-16 will take more memory per character than a single-byte encoding and so is less efficient for displaying English.
UTF-8	Multi-byte - Any	UTF-8 is a multi-byte encoding. It has the advantage that ASCII is preserved. When displaying other languages, UTF-8 combines together multiple bytes to define each code point (character). The efficiency of UTF-8 depends on the language you are trying to display. If you are displaying Western European it will take (on average) 1.3 bytes per character, for Russian 2 bytes (equivalent to UTF-16) but for CJK 3-4 bytes per character.

Encoding	Representation	Description
----------	----------------	-------------

Figure 51 – Common text encodings

What are scripts?

A script is a way of writing a language. It takes the encoding of a language and combines it with its alphabet to render it on screen as a sequence of glyphs. The same language can sometimes be written with more than one script (common among languages in India). Scripts can often be used to write more than one language (common among European languages).

Scripts can be grouped together into four approximate classes. The "small" script class contains a small alphabet with a small set of glyphs to represent each single character. The "large" script class contains a large alphabet and with a larger set of glyphs. The "contextual" script class contains characters that can change appearance depending on their context. And finally the "complex" script class contains characters that are a complex function of the context of the character – there isn't a 1 to 1 mapping between code point and glyph.

Roman	Small script	The Roman encoding has relatively few distinct characters. Each character has a single way of being written. It is written from left to right, top to bottom. Every character has a unique glyph. Characters do not join when written. For example: The quick brown fox.
Chinese	Large script	The Chinese encoding has a large number of distinct characters. Each character has a single way of being written.
Greek	Contextual script	Every character except sigma has a unique glyph. Sigma changes depending on whether it is at the end of a word or not. Characters do not join when written. The text runs left to right, top to bottom. For example: δῖος Ἀχιλλεύς
Arabic	Contextual script	The glyph chosen is dependent on its position in a word. All characters have initial, medial and terminal glyphs. This results in a calligraphic (joined up) style of display. The text runs right to left, top to bottom display. For example: العربية
Devanagari	Complex script	In this script there is no direct mapping from character to glyph. Sequences of glyphs combine depending on their context. The text runs from left to right, top to bottom.

Figure 52 – Common scripts

Introduction to Unicode

Traditionally, computer systems have stored text as 8-bit bytes, with each byte representing a single character (for example, the letter 'A' might be stored as 65). This has the advantage of being very simple and space efficient whilst providing enough (256) different values to represent all the symbols that might be provided on a typewriter. The flaw in this scheme becomes obvious fairly quickly: there are far more than 256 different characters in use in all the writing systems of the world, especially when East Asian ideographic languages are considered. But, in the pre-internet days, this was not a big problem.

LiveCode, as a product first created before the rise of the internet, also adopted the 8-bit character sets of the platforms it ran on (which also meant that each platform used a different character set: MacRoman on Apple devices, CP1252 on Windows and ISO-8859-1 on Linux and Solaris). LiveCode terms these character encodings "native" encodings.

In order to overcome the limitations of 8-bit character sets, the Unicode Consortium was formed. This group aims to assign a unique numerical value ("codepoint") to each symbol used in every written language in use (and in a number that are no longer used!). Unfortunately, this means that a single byte cannot represent any possible character.

The solution to this is to use multiple bytes to encode Unicode characters and there are a number of schemes for doing so. Some of these schemes can be quite complex, requiring a varying number of bytes for each character, depending on its codepoint.

LiveCode previously added support for the UTF-16 encoding for text stored in fields but this could be cumbersome to manipulate as the variable-length aspects of it were not handled transparently and it could only be used in limited contexts. Unicode could not be used in control names, directly in scripts or in many other places where it might be useful.

From LiveCode 7.0, the LiveCode engine has been able to handle Unicode text transparently throughout. The standard text manipulation operations work on Unicode text without any additional effort on your part; Unicode text can now be used to name controls, stacks and other objects; menus containing Unicode selections no longer require tags to be usable - anywhere text is used, Unicode should work.

Creating Unicode Apps

Creating stacks that support Unicode is no more difficult than creating any other stack but there are a few things that should be borne in mind when developing with Unicode. The most important of these is the difference between text and binary data - in previous versions of LiveCode, these could be used interchangeably; doing this with Unicode may not work as you expect (but it will continue to work for non-Unicode text).

When text is treated as binary data (i.e when it is written to a file, process, socket or other object outside of the LiveCode engine) it will lose its Unicode-ness: it will automatically be converted into the platform's 8-bit native character set and any Unicode characters that cannot be correctly represented will be converted into question mark '?' characters.

Similarly, treating binary data as text will interpret it as native text and won't support Unicode. To avoid this loss of data, text should be explicitly encoded into binary data and decoded from binary data at these boundaries - this is done using the `textEncode` and `textDecode` functions (or its equivalents, such as opening a file using a specific encoding).

Unfortunately, the correct text encoding depends on the other programs that will be processing your data and cannot be automatically detected by the LiveCode engine. If in doubt, UTF-8 is often a good choice as it is widely supported by a number of text processing tools and is sometimes considered to be the "default" Unicode encoding.

Things to look out for

- When dealing with binary data, you should use the byte chunk expression rather than `char` - `char` is intended for use with textual data and represents a single graphical character rather than an 8-bit unit.
- Try to avoid hard-coding assumptions based on your native language - the formatting of numbers or the correct direction for text layout, for example. LiveCode provides utilities to assist you with this.
- Regardless of visual direction, text in LiveCode is always in logical order - word 1 is always the first word; it does not depend on whether it appears at the left or the right.
- Even English text can contain Unicode characters - curly quotation marks, long and short dashes,

accents on loanwords, currency symbols...

Using Arrays

For an introduction to arrays, see the section on *Array Variables* in the chapter *Coding in LiveCode*.

When to Use Arrays

Each element in an array can be accessed in constant time. This compares favorably with other functions that become look up information by counting through a variable from the beginning (for example the offset functions). If you consider a problem that requires the computer to search through the data several times then if the computer has to start at the beginning of the variable, the search will get slower and slower as the search function gets further through the data.

Each element in an array can contain data of any length, making it easier to group together records that contain assorted lengths or delimiter characters.

Arrays can contain nested elements. This makes them ideal for representing complex data structures such as trees and XML data that would be hard to represent as a flat structure.

Each sub-array in an array can be accessed and operated on independently. This makes it possible to copy a sub-array to another array, get the keys of a sub-array, or pass a sub-array as a parameter in a function call.

LiveCode includes various functions for converting information to and from arrays, and for performing operations on the contents of arrays.

These characteristics make arrays useful for a number of data processing tasks, including tasks that involve processing or comparing large amounts of data. For example, arrays are ideal if you want to count the number of instances of a specific word in a piece of text. It would be possible to implement such a word count by iterating through each word and checking if it is present in a list of words, then adding a comma followed by a count to that list of words. Such a method is cumbersome to implement and as the list of words gets longer the routine will slow down because LiveCode has to search the list from the start with each new word. Conversely, implementation with an array is simple. Because each element in an array can be named using a text string, we can create an element for each word and add to the element's contents. Not only is the code much shorter but it is also an order of magnitude faster.

```
on mouseUp
  --cycle through each word adding each instance to an array
  repeat for each word tWord in field "sample text"
    add 1 to tWordCount[tWord]
  end repeat
  -- combine the array into text
  combine tWordCount using return and comma
  answer tWordCount
end mouseUp
```

Text in field "Sample Text:

Single Line – execute single line and short scripts
Multiple Lines – execute multiple line scripts
Global Properties – view and edit global properties
Global Variables – view and edit global variables

Resulting value of tWordCount:

Global,4
Line,3
Lines,1
Multiple,2
Properties,2
Single,2

Text in field "Sample Text:	Resulting value of tWordCount:
	Variables,2
	and,3
	edit,2
	execute,2
	scripts,2
	short,1
	view,2
	-,4

Figure 53 – Results of running word count script

Array Functions in LiveCode

The following is a list of all the syntax in LiveCode that works with arrays. For a full description of each one, see the corresponding entry in the LiveCode Dictionary.

Each of these functions can be used on subarrays within an array. Instead of referring to the array variable, refer to x[x]

- **add** adds a value to every element in an array where the element is a number
- **combine** converts an array into text, placing delimiters you specify between the elements
- **customProperties** returns an array of the custom properties of an object
- **delete variable** remove an element from an array
- **divide** divides each element in an array where the element is a number. For example:

divide tArray by 3

Contents of array: Resulting value of tWordCount:

A = 1	0.333333
B = 2	0.666667
C = 3	1
D = 4	1.333333
E = 5	1.666667

Figure 54 – Results of running the divide command on an array

- **element** keyword is used in a repeat loop to loop through every element in an array
- **extents** finds the minimum and maximum row and column numbers of an array whose keys are integers
- **intersect** compares arrays, removing elements from one array if they have no corresponding element in the other
- **keys** returns a list of all the elements within an array
- **matrixMultiply** performs a matrix multiplication on two arrays whose elements are numbers and whose keys are sequential numbers
- **multiply** multiplies a value in every element in an array where the element is a number
- **properties** returns an array of the properties of an object
- **split** convert text to an array using delimiters that you define
- **sum** - returns the sum total of all the elements in an array where the element is a number
- **transpose** swaps the order of the keys in each element in an array whose elements are numbers and whose keys are sequential numbers
- **union** combines two arrays, eliminating duplicate elements

Encoding and Decoding

LiveCode includes a number of built-in functions for encoding and decoding data in a variety of popular formats.

Styled Text

LiveCode supports encoding and decoding styled text as HTML and RTF. This feature is useful when you want to adjust text styles programmatically, or import or export text with style information.

Important: At present HTML conversion support only extends to the styles that the LiveCode field object is capable of rendering.

To convert the contents of a field to HTML compatible tags, use the **HTMLText** property. This property is documented in detail in the LiveCode Dictionary. You can also set this property to display styled text in a field.

Tip: You can get and set the HTMLText property of a chunk within a field, allowing you to view or change text attributes on a section of the text.

For example, to set the text style of line 2 of a field to bold:

```
on mouseUp
  put the htmltext of line 2 of field "sample text" into tText`
  replace "<p>" with "<p><b>" in tText
  replace "</p>" with "</b></p>>" in tText
  set the htmltext of line 2 of field "sample text" to tText`
end mouseUp
```

While this is not as simple as directly applying the style to the text using:

```
set the textStyle of line 2 of field "sample" to "bold"
```

It does allow you to search and replace text styles or perform multiple complex changes to the text based on pattern matching. Performing a series of changes on the HTMLText in a variable then setting the text of a field once can be quicker than updating the style repeatedly directly on the field.

Use the **HTML** keyword with the *Drag and Drop* features and the *Clipboard* features to perform conversion of data to and from HTML when exchanging data with other applications. For more information see the section on *Drag and Drop* in the *Programming a User Interface* guide.

Use the **RTFText** property and **RTF** keyword to work with the RTF format.

Use the **unicodeText** property and **Unicode** keyword to work with Unicode. For more information see the section on *International Text Support*, above.

URLs

To encode and decode URLs, use the **URLEncode** and **URLDecode** functions. The **URLEncode** function will make text safe to use with a URL – for example it will replace *space* with *+*. These functions are particularly useful if you are posting data to a web form using the **POST** command, using the **launch URL** command or sending email using the **revMail** function. For more information see the *LiveCode Dictionary*.

Text:

"Jo Bloggs" \jo@blogs.com\ %22Jo+Bloggs%22+%3Cjo%40blogs.com%3E

URL Encoded result:

Figure 55 – Results of encoding a URL

Binary Data - Base64 (for MIME Email Attachments and Http Transfers)

To encode and decode data in Base64 (e.g. to create an email attachment), use the **base64Encode** and **base64Decode** functions. These functions are useful anywhere you want to convert binary data to text data and back. For more information see the *LiveCode Dictionary*.

Binary Data - Arbitrary Types

Use the **binaryEncode** and **binaryDecode** functions to encode or decode binary data. These functions are documented in detail in the *LiveCode Dictionary*.

Character to Number conversion

To convert a character to and from its corresponding ASCII value use the **charToNum** and **numToChar** functions.

```
put charToNum("a") -- results in 97
```

To convert Unicode characters, set the **useUnicode** local property to true. For more information see the section on *International Text Support*, above.

Data Compression

To compress and decompress data using GZIP, use the **compress** and **decompress** functions.

The following routine asks the user to select a file, then creates a GZip compressed version with a ".gz" extension in the same directory as the original.

```
on mouseUp
  answer file "Select a file:"
  if it is empty then exit mouseUp
  put it & ".gz" into tFileCompressed
  put compress(URL ("binfile:" & it)) into URL ("binfile:" & tFileCompressed)
end mouseUp
```

Encryption

To encrypt or decrypt data use the **encrypt** and **decrypt** commands. These commands are documented in the *LiveCode Dictionary*.

Generating a Checksum

Use the **MD5Digest** to generate a digest of some data. Use this function later to determine if the data was changed or to check that transmission of information was complete.

Tip: In this example we save the MD5Digest of a field when the user opens it for editing. In the field script place:


```

on openField
    set the cDigest of me to md5Digest(the htmlText of me)
end openField

```

If the field is modified (including if a text style is changed anywhere) then a subsequent check of the MD5Digest will return a different result. In the following example we check this digest to determine whether or not to bring up a dialog alerting the user to save changes:

```

on closeStackRequest
    local tDigest
    put md5Digest(the htmlText of field "sample text") into tDigest
    if the cDigest of field "sample text" is not tDigest then
        answer "Save changes before closing?" with "No" or "Yes"
        if it is "Yes" then
            save this stack
        end if
    end if
end closeStackRequest

```

XML

Extensible Markup Language, or XML, is a general-purpose language for exchanging structured data between different applications and across the Internet. It consists of text documents organized into a tree structure. It can generally be understood by both human and machine.

LiveCode includes comprehensive support for XML through its built-in XML library. Additionally, standards exist to support exchange of XML over a network connection (or "web services") – most notably through the XML-RPC and SOAP protocols. LiveCode includes a library for using XML-RPC and there are examples of using LiveCode to build SOAP applications available.

The XML Tree Structure

XML is simply a data tree. It must start with a root node, be well formed and nested. Tags may not overlap. For more information on XML see <http://en.wikipedia.org/wiki/XML>

Figure 56 below shows a typical XML tree. In this example we have represented a simple stack file as XML. The stack file has a single stack with two cards. On the first card there is a field named "Hello" with the contents "Hello World!". There is a second card, which is blank.

Root node	Root element, document element	The start of the XML document, which includes a declaration the file is XML, the version of XML in use and the text encoding
Comment		Comments can be placed anywhere in the tree. They start with \<! and end with ->. They must not contain double dashes --
Node	Element, tag	The items that makes up the content within an XML document

Attributes		Properties attributable to a given node. A node may have zero or more properties
Empty node	Empty element	A method of specifying that a node exists but is empty
Entity reference		A method of specifying special characters. XML includes support for &, \<, >, ' and ". Additional entities can be defined using a Document Type Definition or DTD.

When to use XML

XML has a number of advantages and disadvantages. It is predominantly useful when exchanging data between different applications or systems. However like any method of data storage or transfer it is not suitable for all types of application.

The advantages of XML are: it is text based making it more easily readable by humans as well as just machines; it is self describing; it is based on international standards and in widespread use with a large number of editors available for it; the hierarchical structure makes it suitable for representing many types of document; and it is platform independent.

The disadvantages are that: it is sometimes less efficient than binary or even other forms of text representations of data; for simple applications it is more complicated than may strictly be necessary; and the hierarchical model may not be suitable for all data types.

You may decide that using XML is the best solution for your particular data storage or transmission requirements. Or you may be working on a project with others where using XML or a web service based on it is a requirement. However in many cases a binary format or database will be more appropriate. You should give consideration to the method you intend to use as early as possible in the design of your application.

Methods for Handling XML in LiveCode

LiveCode includes a comprehensive XML library for working with XML documents. Using the XML library has the advantage that we include syntax and functions for performing the common operations on XML that you may need. However the disadvantage is that at present the library is implemented as an external command (included built-in to the LiveCode distribution) and thus does not benefit from native LiveCode-engine syntax. If you have simple XML processing requirements you may prefer to use LiveCode's built in chunk expression support to do the parsing, matching or construction of XML. For more information see the section on *Using Chunk Expressions*. However if you are working with complex XML then the library includes a comprehensive suite of features.

In addition to the XML library, LiveCode has a built-in script-based library for working with XML-RPC.

Tip: To see a list of commands for working with XML-RPC, filter the LiveCode Dictionary with the term XMLRPC.

A lesson demonstrating using the LiveCode XML library is available [here](#)

The XML Library: Loading, Displaying and Unloading XML

This section discusses using the XML library in detail.

Getting Started - Creating an XML Tree in Memory

In order to work with an XML document, you start by creating an XML tree of that document in memory. There are two functions **revXMLCreateTreeFromFile** and **revXMLCreateTree**. Use the former to load XML document from a file and create a tree in memory, use the latter to create an XML tree from

another data source such as a variable, field or download.

```
revXMLCreateTree(XMLText, dontParseBadData, createTree, sendMessages)
revXMLCreateTreeFromFile(filePath, dontParseBadData, createTree, sendMessages)
```

In **revXMLCreateTree** the *XMLText* is the string containing the XML. In **revXMLCreateTreeFromFile** this parameter is replaced with the *filePath* – the file path to the XML document. Both functions return a single value – the ID of the tree that has been created.

Important: Both functions require you to specify all the parameters. You must store the ID returned by these functions in order to access the XML tree later in your script.

The *dontParseBadData* specifies whether or not to attempt to parse poorly formed XML. If this is set to true then bad data will be rejected and generate an error instead of constructing the tree in memory.

The *createTree* specifies whether to create a tree in memory or not. You will generally want this to be true, unless you are intending only to read in an XML file to determine whether or not it is properly structured.

The *sendMessages* specifies whether or not messages should be sent when parsing the XML document. Messages can be useful if you want to implement functionality such as a progress bar, progressively render or progressively process data from a large XML file as it is being parsed. If you set this to true, **revXMLStartTree** will be sent when the parsing starts, **revStartXMLNode** will be sent when a new node is encountered, **revEndXMLNode** will be sent when a node has been completed, **revStartXMLData** will be sent at the start of a new block of data and finally **revXMLEndTree** will be sent when processing is finished.

Retrieving information from an XML tree

Now that you have created your XML tree in memory (above) and stored the tree ID you can use the functions in this section to retrieve information from within the tree.

Important: Any text you fetch using the LiveCode XML library will be in the encoding specified in the root node of the XML tree.

Note: All the examples in this section assume that we have loaded the XML tree depicted in the figure below – XML Tree Representation of a Stack, above. We assume that you have loaded this tree using the **revXMLCreateTree** function described above, and that this function has returned a value of 1 as the ID of the tree.

Retrieving the Root Node

To retrieve the *root node* from your XML tree, use the **revXMLRootNode** function.

```
revXMLRootNode(treeID)
```

The *treeID* contains the ID of the XML tree you want to access. For example, using the following function with sample tree depicted above:

```
put revXMLRootNode(1) into tRootNode
```

Results in *tRootNode* containing: *stackFile*

Retrieving the First Child Element in a Node

To retrieve the first child element use **revXMLFirstChild**.

```
revXMLFirstChild(treeID,parentNode)
```

The *parentNode* contains the path to the node we want to retrieve the first child from. Nodes are referenced using a file-path like format with / used to denote the root and delimit nodes.

We can use this function on our sample XML as follows:

```
-- pass the *stackFile* result in retrieved in tRootNode
-- to the revXMLFirstChild function:
put revXMLFirstChild(1,tRootNode) into tFirstChild
-- EQUIVALENT to the following:
put revXMLFirstChild(1,"stackFile") into tFirstChild
```

This results in *tFirstChild* containing: */stackFile/stack*

Retrieving a list of Children in a Node

To retrieve a list of children of a node use **revXMLChildNames**.

```
revXMLChildNames(treeID, startNode, nameDelim, childName,includeChildCount)
```

The *nameDelim* is the delimiter that separates each name that is returned. To get a list of names, specify return.

The *childName* is the name of the type of children to list.

includeChildCount allows you to include the number of each child in square brackets next to the name.

We can use this function on our sample XML as follows:

```
put revXMLChildNames(1,"/stackFile/stack", return, "card", true) into tNamesList
```

This results in *tNamesList* containing:

```
card[1]
```

```
card[2]
```

Retrieving the Contents of the Children in a Node

To retrieve a list of children of a node including their contents, use **revXMLChildContents**.

```
revXMLChildContents(treeID,startNode,tagDelim,nodeDelim, includeChildCount,depth)
```

See above for an explanation of *treeID*, *startNode* and *tagDelim*.

The *nodeDelim* indicates the delimiter that separates the contents of the node from its name.

The *depth* specifies the number of generations of children to include. If you use -1 as the depth then all children are return.

Using this function on our example XML file as follows:

```
put revXMLChildContents(1, "/stackFile/stack", space, return, true, -1) into tContents
```

This results in *tContents* containing:

card[1]

field[1]

text[1] Hello World!

htmlText[1] \Hello World\

card[2]

Retrieving the Number of Children in a Node

To retrieve the number of children of a node **revXMLNumberOfChildren**.

```
revXMLNumberOfChildren(treeID, startNode, childName, depth)
```

See above for an explanation of *treeID*, *startNode*, *childName* and *depth*.

Using this function on our example XML file as follows:

```
put revXMLNumberOfChildren(1, "/stackFile/stack", "card", -1) into tContents
```

This results in *tContents* containing: 2

Retrieving the Parent of a Node

To retrieve a node's parent use the **revXMLParent** function.

```
revXMLParent(treeID, childNode)
```

See above for an explanation of *treeID* and *startNode*.

Using this function on our example XML file as follows:

```
put revXMLParent(1, "stackFile/stack") into tParent
```

Results in *tParent* containing: /stackFile

Retrieving an Attributes from a Node

To retrieve an attribute from a node use **revXMLAttribute**.

```
revXMLAttribute(treeID,node,attributeName)
```

See above for an explanation of *treeID* and *node*.

The *attributeName* is the name of the attribute you want to retrieve the value for.

Using this function on our example XML file as follows:

```
put revXMLAttribute(1,"/stackFile/stack","rect") into tRect
```

This results in *tRect* containing: 117,109,517,509

Retrieving all Attributes from a Node

To retrieve all attributes from a node use **revXMLAttributes**.

```
revXMLAttributes(treeID,node,valueDelim,attributeDelim)
```

See above for an explanation of *treeID* and *node*.

The *valueDelim* is delimiter that separates the attribute's name from its value.

The *attributeDelim* is delimiter that separates the attribute's name & value pair from each other.

Using this function on our example XML file as follows:

```
put revXMLAttributes(1, "/stackFile/stack/card/field", tab, return) into  
tFieldAttributes
```

This results in *tFieldAttributes* containing:

name Hello

rect 100,100,200,125

Retrieving the Contents of Attributes

To retrieve the contents of a specified attribute from a node and its children, use **revXMLAttributeValues**.

```
revXMLAttributeValues(treeID, startNode, childName, attributeName, delimiter,  
depth)
```

See above for an explanation of *treeID*, *startNode* and *depth*.

The *childName* is the name of the type of child to be searched. Leave this blank to include all types of

children.

The *attributeName* is the name of the attribute to return the values for.

The *delimiter* is the delimiter to be used to separate the values returned.

Using this function on our example XML file as follows:

```
put revXMLAttributeValues(1, "/stackFile/", , "rect", return, -1) into tRectsList
```

This results in *tRectsList* containing:

117,109,517,509

100,100,200,125

Retrieving the Contents of a Node

To retrieve the contents of a specified node, use **revXMLNodeContents**.

```
revXMLNodeContents(treeID, node)
```

See above for an explanation of *treeID* and *node*.

Using this function on our example XML file as follows:

```
put revXMLNodeContents(1, "/stackFile/stack/card/field/htmlText") into tFieldContents
```

This results in *tFieldContents* containing:

\Hello World\

The entity references for the \< and > symbols have been translated into text in this result.

Retrieving Siblings

To retrieve the contents of the siblings of a node, use **revXMLNextSibling** and **revXMLPreviousSibling**.

```
revXMLNextSibling(treeID,siblingNode)
revXMLPreviousSibling(treeID,siblingNode)
```

The *siblingNode* is the path to the node to retrieve the siblings from.

Using this function on our example XML file as follows:

```
put revXMLPreviousSibling(1, "/stackFile/stack/card[2]") into tPrev
put revXMLNextSibling(1, "/stackFile/stack/card") into tNext
```

This results in *tPrev* containing:

```
/stackFile/stack/card[1]
```

And *tNext* containing:

```
/stackFile/stack/card[2]
```

Searching for a Node

To search for a node based on an attribute, use **revXMLMatchingNode**.

```
revXMLMatchingNode(treeID, startNode, childName, \ attributeName, attributeValue,
depth, [caseSensitive])
```

See above for an explanation of *treeID*, *startNode* and *depth*.

The *childName* is the name of the children you want to include in the search. If you leave this blank all children are searched.

The *attributeName* is the name of the attribute you want to search.

attributeValue is the search term you want to match.

caseSensitive optionally specifies whether the search should be case sensitive. The default is false.

Using this function on our example XML file as follows:

```
put revXMLMatchingNode(106,"/", , "name", "Hello", -1) into tMatch
```

This results in *tMatch* containing:

```
/stackFile/stack/card[1]/field
```

Retrieving an Outline of the Tree (or Portion Thereof)

To retrieve the contents of a specified node, use **revXMLTree**.

```
revXMLTree(treeID, startNode, nodeDelim, padding, includeChildCount, depth)
```

See above for an explanation of *treeID*, *startNode*, *includeChildCount* and *depth*.

The *nodeDelim* is the delimiter that separates each node in the tree. Use return to retrieve a list of nodes.

padding is the character to use to indent each level in the tree.

Using this function on our example XML file as follows:

```
put revXMLTree(106,"/",return,space,true,-1) into tTree
```

This results in *tTree* containing:


```

stackFile[1]
stack[1]
card[1]
field[1]
text[1]
htmlText[1]
card[2]

```

Retrieving the Tree as XML (or Portion Thereof)

To retrieve the tree as XML use **revXMLText**. Use this function to save the XML to a file after modifying it.

```
revXMLText(treeID, startNode, [formatTree])
```

See above for an explanation of *treeID* and *startNode*.

The *formatTree* specifies whether or not to format the returned tree with return and space characters to make it easier to read by a human.

Using this function on our example XML file as follows:

```

ask file "Save XML as:"
put revXMLText(106,"/",true) into URL ("file:" & it)

```

This results in the file the *user specifies* containing:

```

<stackFile>

<stack name="Example" rect="117,109,517,509">

  <card>

    <field name="Hello" rect="100,100,200,125">

      <text>Hello World!</text>

      <htmlText>&lt;&gt;Hello World&lt;/p&gt;</htmlText>

    </field>

  </card>

</stack>

</stackFile>

```

Validating against a DTD

To check the syntax of an XML file conforms to a DTD use **revXMLValidateDTD**. For more information on this function, see the *LiveCode Dictionary*.

Listing all XML Trees in Memory

To generate a list of all XML trees in memory, use **revXMLTrees**. For more information on this function, see the *LiveCode Dictionary*.

Removing an XML Tree from Memory

To remove an XML tree from memory, use **revXMLDeleteTree**. To remove all XML trees from memory, use **revXMLDeleteAllTrees**. Both functions take a single parameter – the ID of the tree to be deleted. You should delete a tree when you have stopped using it. For more information on these functions, see the *LiveCode Dictionary*.

Once an XML tree has been removed from memory, there is no way to get it back. Use the **revXMLText** function to retrieve the contents of the entire tree and save it first.

The XML Library: Editing XML

This section discusses how to edit XML trees. Before reading this section you should read the section above on loading, displaying and unloading XML.

Adding a new Child Node

To add a new node use the **revXMLAddNode** command.

```
revXMLAddNode treeID, parentNode, nodeName, nodeContents, [location]
```

See above for an explanation of *treeID*.

The *parentNode* is the name of the node you want to add the child to.

The *nodeName* is the name of the new node to create.

nodeContents is the contents of the new node.

location - optionally specify "before" to place the new child at the start of the child nodes.

Use this function to add a button to our example XML file as follows:

```
revXMLAddNode 1, "/stackFile/stack/card/", "button", ""
```

This results in our tree containing a new button:

```

<?xml version="1.0" encoding="UTF-8"?>

<!-- This is a comment. -->

<stackFile>

<stack name="Example" rect="117,109,517,509">

<card>

<field name="Hello" rect="100,100,200,125">

<text>Hello World!</text>

<htmlText>&lt;p&gt;Hello World&lt;/p&gt;</htmlText>

</field>

<button></button>

</card>

</stack>

</stackFile>

```

To create another node at the same level as another node, use the **revXMLInsertNode** command instead.

Appending XML to a tree

To add a new node use the **revXMLAppend** command.

```
revXMLAppend treeID, parentNode, newXML
```

See above for an explanation of *treeID* and *parentNode*.

The *newXML* is XML you want to append to the tree.

Moving, Copying or Deleting a Node

To move a node use the **revXMLMoveNode** command.

```
revXMLMoveNode treeID, sourceNode, destinationNode [, location] [, relationship]
```

See above for an explanation of *treeID*.

The *sourceNode* is the path to the node you want to move.

The *destinationNode* is the path to the node you to move to.

The *location* specifies where the node should be moved to in the list of siblings – it can be either "before" or "after".

The *relationship* allows you to specify whether to place the node alongside the destination as a sibling or below the destination as a child.

To copy a node use **revXMLCopyNode**.

To delete a node use **revXMLDeleteNode**.

Putting data into a Node

To put data into a node use the **revXMLPutIntoNode** command.

```
revXMLPutIntoNode treeID,node,newContents
```

See above for an explanation of *treeID* and *node*.

The *newContents* is the text that the new node will contain.

Setting an Attribute

To set an attribute use the **revXMLSetAttribute** command.

```
revXMLSetAttribute treeID,node,attributeName,newValue
```

See above for an explanation of *treeID* and *node*.

The *attributeName* is the name of the attribute you want to set the attribute on.

The *newValue* is the value to set for the attribute.

Using this function to add a "showBorder" property to our field:

```
revXMLSetAttribute 1, "/stackFile/stack/card/button", "showBorder","true"
```

The field tag in our tree now looks like this:

```
<field name="Hello" rect="100,100,200,125" showBorder="true">
```

Adding a DTD

To add a DTD to the tree, use the **revXMLAddDTD** command.

```
revXMLAddDTD treeID,DTDText
```

See above for an explanation of *treeID*.

The *DTDText* is the text of the DTD to add.

Sorting

Sorting data is a common and fundamental operation. Sorting allows you to display data in a user-friendly fashion or code a number of algorithms. LiveCode's intuitive sort features give you the power and flexibility to perform any kind of sorting you may require.

The Sort Container Command: Overview

To sort data, use the **sort container** command.

```
sort [{lines | items} of] container [direction] [sortType] [by sortKey]
```

The *container* is a field, button, or variable, or the message box.

The *direction* is either ascending or descending. If you don't specify a direction, the sort is ascending.

The *sortType* is one of text, numeric, or dateTime. If you don't specify a sortType, the sort is by text.

The *sortKey* is an expression that evaluates to a value for each line or item in the container. If the *sortKey* contains a chunk expression, the keyword *each* indicates that the chunk expression is evaluated for *each* line or item. If you don't specify a sortKey, the entire line (or item) is used as the sortKey.

The following example sorts the *lines* of a variable alphabetically:

```
sort lines of field "sample text" ascending text
sort lines of tText descending text
```

The following example sorts a collection of *items* numerically:

```
sort items of field "sample csv" ascending numeric
sort items of tItems descending numeric
```

The Sort Container Command: Using Sort Keys

The *sortKey* syntax allows you to sort each line or item based on the results of an evaluation performed on each line or item.

To sort the lines of a container by a specific item in each line:

```
sort lines of tContainer by the first item of each
sort lines of tContainer by item 3 of each
```

The *sortKey* expression will only be evaluated once for every element that is to be sorted. This syntax allows a variety of more complex sort operations to be performed.

The following example will extract the minimum and maximum integers present in a list:

```

set the itemDelimiter to "."
sort lines of fld 1 numeric by char 2 to -1 of the first item of each
put char 2 to -1 of the first item \
  of the first line of fld 1 into tMinimumInteger
put char 2 to -1 of the first item \
  of the last line of fld 1 into tMaximumInteger

```

Original list:	Result:
F54.mov	tMinimumInteger is 3
M27.mov	tMaximumInteger is 54
M7.mov	
F3.mov	

Figure 57 – Results of sort command using sort key

The Sort Container Command: Sorting Randomly

To sort randomly, use the **random** function to generate a random number as the *sortKey* for each line or item, instead of evaluating the line or item's contents. For example:

```

put the number of lines of tExampleList into tElementCount
sort lines of tExampleList ascending numeric by random(tElementCount)

```

The Sort Container Command: Stable Sorts - Sorting on Multiple Keys

To sort a list by multiple criteria you can sort multiple times. This is because LiveCode uses a stable sort, meaning that if two items have the same sort key their relative order in the output will not change. To perform a stable sort, start with the least significant or important criteria and work up to the most important or significant. For example:

```

sort lines of fld 1 ascending numeric by item 2 of each
sort lines of fld 1 ascending text by the first item of each

```

Original list:	Result:
Oliver,1.54	Elanor,5.67
Elanor,5.67	Elanor,6.3
Marcus,8.99	Marcus,8.99
Elanor,6.34	Oliver,1.54
Oliver,8.99	Oliver,8.99
Tim,3.44	Tim,3.44

Figure 58 – Results of sorting multiple items

Tip: If you have a large data set and want to improve performance by only performing a single sort, you can construct a sort key that gives the appropriate ordering. In this example a good way to do that is to use the **format** function to construct a fixed length string, one element per sort:

```
sort lines of fld 1 ascending text by \\  
format("%-16s%08.2f", item 1 of each, item 2 of each)
```

This formats each individual line similar to the following:

```
Oliver 00001.54  
Elanor 00005.67
```

These lines now sort the required way as if the first field (the name) ties, the order is determined by the second field – due to the use of padding characters making all the fields the same size.

Sorting Cards

To sort cards, use the **sort** command.

```
sort [marked] cards [of stack] [direction] [sortType] by sortKey
```

The *stack* is a reference to any open stack. If you don't specify a *stack*, the cards of the current stack are sorted.

The *direction* is either ascending or descending. If you don't specify a *direction*, the sort is ascending.

The *sortType* is one of text, international, numeric, or dateTime. If you don't specify a *sortType*, the sort is by text.

The *sortKey* is an expression that evaluates to a value for each card in the stack. Any object references within the *sortKey* are treated as pertaining to each card being evaluated, so for example, a reference to a field is evaluated according to that field's contents on each card. Typically the sort command is used with *background* fields that have their *sharedText* property set to false so that they contain a different value on each card.

For example to sort cards by the contents of the last name field on each:

```
sort cards by field "Last Name"
```

To sort cards by the numeric value in a ZIP Code:

```
sort cards numeric by field "ZIP code"
```

Tip: To sort cards by a custom expression that performs a calculation, you can create a custom function:

```
sort cards by myFunction() -- uses function below

function myFunction
    put the number of buttons of this card into tValue
    -- perform any calculation on tValue here
    return tValue
    -- sort will use this value
end myFunction
```


Transferring Information with Files, the Internet and Sockets

Introduction

Reading and writing data to files or transferring data over the Internet are important functions in most applications. LiveCode provides a rich feature set for performing these operations.

Accessing data from a file typically takes just a single line of code. LiveCode's file path syntax uses the same format on each platform so you typically don't have to rewrite your file handling routines to deploy cross platform. A set of functions provides for copying, deleting or renaming files, as well as accessing appropriate system and user folders.

LiveCode includes functions for downloading and uploading data to the Internet. Simple downloads and uploads can be performed with just a single line of code. Support for the http, ftp and post protocols is included. Syntax is included that allows downloading in both the foreground and background. Additional library commands allow you to construct multipart form data, send ftp commands and more.

LiveCode includes built-in support for https, SSL & encryption.

If the built-in protocol support doesn't do what you need, LiveCode also allows you to implement your own Internet protocols using its straightforward socket support. A very basic client server application can be written in a few lines of code.

File Name Specifications and File Paths

A file path is a way of describing the location of a file or folder so that it can be found by a handler. File paths are used throughout LiveCode: when you read to and write from text files, when you reference an external video file to display in a player, and in many other situations. If your application refers to external files in any way, an understanding of file path is essential.

This topic discusses the syntax for creating and reading a file reference, and how to relate file paths to the location of your application so that they'll be accessible when your application is installed on another system with a different folder structure.

What is a File Path?

A file path is a description of the exact location of a file or folder. The file path is created by starting at the top of the computer's file system, naming the disk or volume that the file is on, then naming every folder that encloses the file, in descending order, until the file is reached.

Locating a file

For example, suppose you want to describe the location of a file called "My File", which is located inside a folder called "My Folder". That folder is in turn located inside a folder called "Top Folder", which is on a drive called "Hard Disk". You need all this information to completely describe where the file is: - Hard Disk - Top Folder - My Folder - My File

If someone tells you which disk the file is on, then which folder to open, and so on, you can find the file by opening each successive icon on your computer's desktop. By starting with the disk, then opening each enclosing folder in succession until you arrive at the file, you can find exactly the file that's being described.

The structure of a file path

A file path specifies each level of the hierarchy that encloses the file. LiveCode presents the information in a file path that might look like this:

```
/Hard Disk/Top Folder/My Folder/My File
```

You can see that to write a file path, you start by naming the disk the file is on, then add each enclosing folder in order until you arrive at the file.

To see the path to a file, enter the following in the message box:

```
answer file "Choose a file: "; put it
```

This displays the file path for the file you choose.

Important: Each platform has its own way for programmers to specify file paths. The file path shown above is in the usual style for file paths on Linux systems. For cross-platform compatibility, LiveCode uses this same forward slash / character in its file path regardless of the current platform. This way, you can generally specify file and work with paths in your scripts without having to convert them when you switch platforms.

File paths on Windows systems

On Windows systems, disks are named with a drive letter followed by a colon character (:). A typical LiveCode file path on a Windows system looks like this:

```
C:/folder/file.txt
```

File paths on OS X systems

On OS X systems, the startup disk, rather than the desktop, is used as the top level of the folder structure. This means that the startup disk's name does not appear in file paths. Instead, the first part of the file path is the top-level folder that the file is in.

If the disk "Hard Disk" is the startup disk, a typical path on OS X systems might look like this:

```
/Top Folder/My Folder/My File
```

Notice that the disk name isn't part of this path.

Note: If you need to find out the startup disk's name, check the first disk name returned by the **volumes** function.

For files on a disk that isn't the startup disk, the file path starts with "/Volumes" instead of "/". A typical file path to a file that's on a non-startup disk on an OS X system looks like this:

```
/Volumes/Swap Disk/Folder/file.txt
```

Folder paths

You construct the path of a folder the same way as the path to a file. A folder path always ends with a slash character (/). This final slash indicates that the path is to a folder rather than a file.

For example, this pathname describes a folder called "Project" inside a folder called "Forbin" on a disk named "Doomsday":

```
/Doomsday/Forbin/Project/
```

If "Project" is a file, its pathname looks like this, without the final slash:

```
/Doomsday/Forbin/Project
```

File paths for OS X bundles

A bundle is a special type of folder, used on OS X, that is presented to the user as a single file but that is maintained internally by the operating system as a folder. Many OS X applications – including LiveCode and the applications it creates – are stored and distributed as bundles that contain several files. When the user double-clicks the bundle the application starts up instead of a folder window opening to show the bundle's contents.

You can take advantage of the bundle concept to include any needed support files with your application. If you place the files in the application's bundle, users ordinarily never see them, and the entire application-support files and all--behaves as a single icon.

Tip: To see the contents of a bundle, right-click (or control click) the bundle and choose "Show Package Contents" from the contextual menu.

Most of the time, the distinction between bundles and files doesn't matter. However we recommend that you treat them as files when working from the perspective of a user but otherwise refer to them as folders when coding. This will help to keep your code readable. Thus if you are selecting a bundle in a file dialog use the answer file form. When moving or renaming a bundle, refer to them as a folder.

Moving, renaming, or deleting a bundle

When using the **rename** command, to rename a bundle, use the rename folder form of the command:

```
rename folder "/Volumes/Disk/Applications/MyApp/" to \
"/Volumes/Disk/Applications/OtherApp/"
```

Similarly, when dealing with a bundle, use the **delete folder** command instead of **delete file**, and the **revCopyFolder** command instead of **revCopyFile**.

Referring to files inside a bundle

When referring to a file that's inside a bundle, you can treat the bundle just as if it were a folder. For example, if you have placed a file called "My Support.txt" inside your application's bundle, the absolute path to the file might look like this:

```
/Volumes/Disk/Applications/MyApp/My Support.txt
```

The / character in a file or folder name

The slash (/) is not a legal character in Unix or Windows file or folder names, but it is legal for Mac OS X file or folder names to contain a slash. Since a slash in a file or folder name would cause ambiguity – is the slash part of a name, or does it separate one level of the hierarchy from the next? – LiveCode substitutes a colon (:) for any slashes in folder or file names on Mac OS X systems.

For example, if a file on a Mac OS X system is named "Notes from 12/21/93", you refer to it in a script as "Notes from 12:21:93". Since the colon is not a legal character in Mac OS X folder or file names, this removes the ambiguity.

Absolute and Relative File Paths

When describing how to get to a file, you have two options. You can start from the top level, the name of the disk, and name each of the enclosing folders until you get to the file. This is called an absolute path, because it's independent of where you start from. Or you can start from the current folder and describe how to get to the file from there. This is called a relative path, because it depends on where you start.

All the file paths shown so far in this topic are absolute paths.

Absolute file paths

Absolute file paths do not depend on which folder your stack file is in or on where the current folder is. An absolute path to a particular folder or file is always written the same way.

For example, suppose your application is in a folder called "Application Folder", and you want to specify a file called "Westwind" which is in a folder called "Stories" inside "Application Folder". - Hard Disk - Top Folder - My Folder - My File - Application Folder - My Application - Stories - Westwind

The absolute file path of your application looks like this:

```
/Hard Disk/Application Folder/My Application
```

and the absolute path of the "Westwind" file looks like this:

```
/Hard Disk/Application Folder/Stories/Westwind
```

Note: On Mac OS X, and Linux systems, absolute file paths always start with a slash character. On Windows systems, absolute file paths always start with a drive letter followed by a colon (:).

Relative file paths

Now suppose you want to tell someone how to get to the "Westwind" file, starting from the folder containing the application.

Since the application is in "Application Folder", we don't need to include the steps to get to "Application Folder". Instead, we can describe the location of the "Westwind" file with this relative pathname:

Stories/Westwind

This relative pathname starts at "Application Folder"--the folder that holds the application--and describes how to get to the "Westwind" file from there: you open the folder "Stories", then find "Westwind" inside it.

A relative file path starts at a particular folder, rather than at the top of the file system like an absolute file path. The relative file path builds a file path from the starting folder to the file or folder whose location is being specified.

Finding the current folder

By default, the current folder is set to the folder containing the application (either the LiveCode development environment or your application, depending on whether your application is a standalone). So in the example above, the current folder is "Application Folder", because that's where the running application is located.

Note: To change the current folder, set the **defaultFolder** property.

Going up to the parent folder

The relative path ".." indicates the current folder's parent folder. If the current folder is "Stories", the relative path

```
..
```

means the same thing as the absolute path

```
/Hard Disk/Application Folder/
```

Going up multiple levels

To go up more than one level, use more than one "../". To go up two levels, use "../.."; to go up three levels, use "../../..", and so forth.

For example, suppose the current folder is "Stories", and its absolute path looks like this:

```
/Hard Disk/Application Folder/Stories/
```

To get to "My Application" in "Application Folder", you go up one level to "Application Folder", then down one level to "My Application". The relative path looks like this:

```
../My Application
```

To get to "Top Folder" on "Hard Disk", you go up two levels--to "Application Folder", then to "Hard Disk"--and then down one level to "Top Folder". The relative path looks like this:

```
../../Top Folder/
```

Starting at the home directory

On OS X and Unix systems, the "~" character designates a user's home directory.

A path that starts with "~/\" is a relative path starting with the current user's home directory. A path that starts with "~\", followed by the user ID of a user on that system, is a relative path starting with that user's home directory.

When to use relative and absolute file paths

Absolute file paths and relative file paths are interchangeable. Which one to use depends on a couple of factors.

Absolute file paths are easy to understand and they don't change depending on the current folder. This can be an advantage if you are changing the defaultFolder regularly.

However absolute file paths always include the full name of the hard disk and folders leading up to the current working folder. Therefore, if you plan to distribute your application you will want to work with relative paths, so that media shipped in subfolders with your application is still easy to locate.

Tip: By default, when linking to an image or resource using the Inspector, LiveCode inserts an absolute file path. If you plan to distribute your application, locate your media in a subfolder next to the stack you are working on and convert these file paths to relative file paths by deleting the directories up to the one you are working in. This will mean you don't need to make any changes when it comes time to distribute your application.

It's OK to use absolute paths to specify files or folders that the user selects after installation. For example, if you ask the user to select a file (using the **answer file** command) and read data from the file, there's no need to convert the absolute path that the **answer file** command provides to a relative path. Because you're using the path right after you get it from the **answer** command, you know that the disk name and folder structure aren't going to change between getting the path and using it.

Special Folders

Modern operating systems each have a set of special-purpose folders designated for a variety of purposes. If you are writing an application it is recommended that you make use of these folders where appropriate so that you provide the best possible user experience. For example, the contents of the desktop reside in a special folder; there is a folder set aside for fonts; there is a folder for application preferences; and so on.

These special folders don't always have the same name and location, so you can't rely on a stored file path to locate them. For example, if your application is installed onto an OS localized into a different language, the names of the file path will be different, on some Windows special folders are named or placed differently depending on what version of Windows is running, etc.

To find out the name and location of a special folder, regardless of any of these factors, you use the **specialFolderPath** function. The function supports a number of forms for each operating system, describing the special folders for each one. Some of the forms are the same cross-platform. The following example will get the location of the Desktop folder on Windows, Mac OS X or Linux:

```
put specialFolderPath("Desktop") into myPath
```

To get the path to the Start menu's folder on a Windows system:

```
put specialFolderPath("Start") into myPath
```

For a complete list of possible folders see the, **specialFolderPath** in the *LiveCode Dictionary*.

File Types, Application Signatures & File Ownership

When you double-click a document file, it automatically opens in the application it's associated with. Each operating system has a different method for associating files with an application. In order to create files that belong to your standalone application, you need to set up the association appropriately for each platform you distribute on.

This topic describes how to correctly associate your application with the files it creates.

Windows File Extensions and Ownership

When a file is saved on a Windows system, a three-character extension is usually added to the file's name. The extension specifies the format of the file.

To determine which application to launch when the user double-clicks a file, Windows checks the Windows registry to find out what application has registered itself as owning the file's extension. Each application can add keys to the registry to identify certain file extensions as belonging to it.

Applications that don't own files

If your application does not create files that you want the application to own, you don't need to make any modifications to the registry or specify any extensions.

Applications that own their own files

If your application creates files with its own custom extension, when you install the application, you should make changes to the Windows registry to identify the extension as belonging to your application.

Popular Windows installer programs will make these registry changes automatically for you. You can also perform these registry changes using the **setRegistry** function.

Installing custom icons

Each Windows file can display its own icon. You can have separate icons for your application and for files it owns. Icon files must be stored in .ico format.

Custom application icons

If you want to include a custom icon for your application, use the "Application Icon" option on the Windows screen of the Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application. For more information, see the chapter on *Deploying Your Application*.

Custom file icons

To include a custom icon for your documents, use the "Document Icon" option on the Windows screen of the Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application.

Important: For the correct icon to appear on files your application creates, the file's extension must be registered in the Windows registry.

File extensions

You can add an extension to the name of any Windows file. The extension may contain letters A-Z, digits 0-9, ' (single quote), !, @, #, \$, %, ^, &, (,), -, _, {, }, `, or ~.

The Windows registry associates applications with the extension for the files they own.

OS X File Types and Creators

On OS X each file has a file extension which determines which application owns it. However OS X systems can also use the unique four-character creator signature and a four-character file type (see below for more information).

OS X applications store file association information in a property list file, or plist. Each application's plist is stored as part of its application bundle.

Applications that don't own files

To assign your unique creator signature when building an application, enter the signature on the OS X screen of the Standalone Application Settings window. LiveCode automatically includes the creator signature in the application's plist.

Applications that own their own files

If your application creates files with your application's creator signature, you should include in your application's plist an entry for each file type you use. Once you have built your standalone application, follow these steps to open the plist file:

1. Right click on your application bundle, navigate to the contents folder and open the "Info.plist" file. If you have installed Apple's developer tools, you have an application called "Property List Editor", which you can use to make changes to the plist file. Otherwise, you can edit the file in a text editor.
2. Locate the information for the document type. In Property List Editor, expand the "Root" node, then expand the "CFBundleDocumentTypes" node, then expand the "0" node. In a text editor, locate "CFBundleDocumentTypes". Below it, note the tags `<array>` and `<dict>`. The information for the first document type is between `<dict>` and `</dict>`.
3. Enter the file description, which is a short phrase describing what kind of file this is. In Property List Editor, change the value of "CFBundleTypeName" to the description you want to use. In a text editor, locate "CFBundleTypeName" in the document information. Below it is the file description, enclosed between `<string>` and `</string>`:

```
<string>LiveCode Stack</string>
```

Change the description to the one you want to use.

Do not change the tags (enclosed in "<" and ">"). Only change what's between them.

4. Enter the file extension. In Property List Editor, expand "CFBundleTypeExtensions" and enter the file extension in the "0" node. In a text editor, locate "CFBundleTypeExtensions" in the document information. Below it is the extension, enclosed in `<array>` and `<string>` tags. Change the extension to the one you want to use.

5. Enter the four-character file type. In Property List Editor, expand "CFBundleTypeOSTypes" and enter the file type in the "0" node. In a text editor, locate "CFBundleTypeOSTypes" in the document information. Below it is the file type, enclosed in `<array>` and `<string>` tags. Change the file type to the one you want to use.

If the format for this type of file is standard (such as plain text), use a standard type (such as "TEXT"). If the format belongs to your application, use a custom file type of your choice.

Important: Apple reserves all file types with no uppercase letters. If you use a custom file type for your application, make sure it contains at least one uppercase letter.

If you want to assign more file types to your application, copy the portion of the plist file inside the "CFBundleTypes" array between `<dict>` and `<dict>`, including these tags. The "CFBundleTypes" node should now contain two `<dict>` nodes and all their contents. Repeat the steps above for each different file type your application can create.

Creating Files

When your application creates files, set the **fileType** property to the desired creator signature and file type for the new file. (For stack files created with the **save** command, use the **stackFileType** property instead.) When creating files, the application uses the current value of the **fileType** or **stackFileType** property to determine what creator and file type the new file should have.

It's important to understand that a file's creator signature determines which application is launched automatically when you double-click the file, but doesn't prevent other applications from being able to open that file. For example, if your application creates files of type "TEXT", any text editor can open the files. If your application creates stack files, and uses the file type "RSTK", then LiveCode will be able to open the stack files, as well as your application.

File extensions

You can add an extension to the name of any OS X file. When the user double-clicks a file with no creator signature, the operating system uses the extension to determine which application to use to open the file.

An application bundle's name should end with the extension ".app".

Note: Apple's recommendations for determining file type and creator on OS X systems are currently in flux. The recommended method for the present is to set a file type and creator signature, and also attach an extension to the end of each file's name when it is created. Valid extensions on OS X systems are up to twelve characters in length, and may include the letters a-z, the digits 0-9, \$, %, _, or ~. For up-to-date information on Apple's recommendations for OS X, see Apple's [developer documentation](#).

Mac OS X Classic File Types and Creators

When a file is saved on a Mac OS X system, a four-character creator signature is saved with it. The creator signature specifies which application owns the file. Every Mac OS X application should have a unique creator signature. (Apple maintains a registry of creator signatures on its [web site](#)).

Applications that don't own files

To assign your unique creator signature when building an application, enter the signature on the Mac OS X screen of the Standalone Application Settings window. LiveCode automatically includes the resources needed for Mac OS X to recognize the creator signature.

Applications that own their own files

If your application creates files with your application's creator signature, you should include in your application a set of resources for each file type you use. Once you have saved your standalone application, open the application file in ResEdit and follow these steps:

1. Open the BNDL resource window, then open the BNDL 128 resource. The BNDL 128 resource contains a single entry ("APPL").
2. Choose "Create New File Type" from the Resources menu. A new entry appears below the "APPL" entry.
3. In the Type field, enter the four-character file type. If the format for this type of file is standard (such as plain text), use a standard type (such as "TEXT"). If the format belongs to your application, use a custom file type of your choice.

Repeat steps 2-3 for each different file type your application can create.

When your application creates files, set the **fileType** property to the desired creator signature and file type for the new file. For stack files created with the save command, use the **stackFileType** property instead. When creating files, the application uses the current value of the **fileType** or **stackFileType** property to determine what creator and file type the new file should have.

Installing custom icons

Each Mac OS **file** may display any of six different icons, depending on context and on the number of colors the screen can display: large (32x32 pixel) icons and small (16x16 pixel) icons, each in black-and-white, 16 colors, and 256 colors.

Mac OS provides default icons that are used for applications and documents that don't have their own. If you want your application or the documents it owns to display a custom icon, you must create the icons and then attach them to the application.

Custom application icons

If you want to include a custom icon for your application, use ResEdit or a similar tool to create a set of icon resources. There are six standard icon resource types: ICN# (black-and-white), icl4 (four-bit color), icl8 (8-bit color), ics# (black-and-white small), ics4 (4-bit small), and ics8 (8-bit small). Each of these application icons should have the resource ID 128.

Save the icons in a single file, and use the "Include resources from file" option on the Mac OS X screen of the Standalone Application Settings window to specify the file. When you build the application, the icons will be included in the application's file.

Linux File Extensions

Linux systems do not have an overall required method for specifying a file's type, but most files on a Linux system are created with extensions in the file name, similar to the extensions used on Windows systems. These extensions may be of any length and may include any characters (other than /).

Working with URLs

A URL is a container for a file (or other resource), which may either be on the same system the application is running on, or on another system that's accessible via the Internet.

This topic discusses the various URL schemes that LiveCode implements, how to create and manipulate files using URLs, and how to transfer data between your system and an FTP or HTTP server.

To fully understand this topic, you should know how to create objects and write short scripts, and understand how to use variables to hold data. You should also have a basic understanding of how the Internet works.

An Overview of URLs

In the LiveCode language, a URL is a container for a **file** or other document, such as the output of a CGI on a web server. The data in a URL may be on the same system the application is running on, or may be on another system.

URLs in LiveCode are written like the URLs you see in a browser. You use the **URL** keyword to designate a URL, enclosing the URL's name in double quotes:

```
put field "Info" into URL "file:myfile.txt"
get URL "http://www.example.org/stuff/nonsense.html"
put URL "ftp://ftp.example.net/myfile" into field "Data"
```

URL Schemes

A URL scheme is a type of URL. LiveCode supports five URL schemes with the **URL** keyword: **http**, **ftp**, **file**, **binfile**, and (for backwards compatibility on Mac OS X) **resfile**.

The **http** and **ftp** schemes designate documents or directories that are located on another system that's accessible via the Internet. The **file**, **binfile**, and **resfile** schemes designate local files.

The http scheme

An **http** URL designates a document from a web server:

```
put URL "http://www.example.org/home.htm" into field "Page"
```

When you use an **http** URL in an expression, LiveCode downloads the URL from the server and substitutes the downloaded data for the URL.

When you put something into an **http** URL, LiveCode uploads the data to the web server:

```
put field "Info" into URL "http://www.example.net/info.htm"
```

Note: Because most web servers do not allow **http** uploads, putting something into an **http** URL usually will not be successful. Check with the server's administrator to find out whether you can use the **http** protocol to upload files.

For more details about **http** URLs, see the entry for the **http** keyword in the LiveCode Dictionary.

The ftp scheme

An **ftp** URL designates a file or directory on an FTP server:

```
get URL "ftp://user:passwd@ftp.example.net/picture.jpg"
```

When you use an **ftp** URL in an expression, LiveCode downloads the URL from the server and

substitutes the downloaded data for the URL. When you put something into an ftp URL, LiveCode uploads the data to the ftp server:

```
put image 10 into URL \
    "ftp://user:passwd@ftp.example.net/picture.jpg"
```

FTP servers require a user name and password, which you can specify in the URL. If you don't specify a user name and password, LiveCode adds the "anonymous" user name and a dummy password automatically, in accordance with the convention for public FTP servers.

Note: Uploading to an FTP server usually requires a registered user name and password.

For more details about ftp URLs, see the entry for the ftp keyword in the LiveCode Dictionary.

Directories on an FTP server

A URL that ends with a slash (/) designates a directory (rather than a file). An ftp URL to a directory evaluates to a listing of the directory's contents.

The file scheme

A file URL designates a file on your system:

```
put field "Stuff" into URL "file:/Disk/Folder/testfile"
```

When you use a file URL in an expression, LiveCode gets the contents of the file you designate and substitutes it for the URL. The following example puts the contents of a file into a variable:

```
put URL "file:myfile.txt" into myVariable
```

When you put data into a file URL, LiveCode puts the data into the file:

```
put myVariable into URL "file:/Volumes/Backup/data"
```

Note: As with local variables, if the file doesn't exist, putting data into it creates the file.

To create a URL from a file path that LiveCode provides, use the & operator:

```
answer file "Please choose a file to get:"
get URL ("file:" & it)
```

File path syntax and the file scheme:

The file URL scheme uses the same file path syntax used elsewhere in LiveCode statements. You can use both absolute paths and relative paths in a file URL.

Conversion of end-of-line markers

Different operating systems use different characters to mark the end of a line. Mac OS X uses a return character (ASCII 13), Linux systems use a linefeed character (ASCII 10), and Windows systems use a return followed by a linefeed. To avoid problems when transporting a stack between platforms, LiveCode always uses linefeeds internally when you use a `file` URL as a container. LiveCode translates as needed between the your system's end-of-line marker and LiveCode's linefeed character. To avoid this translation, use the `binfile` scheme (see below).

The binfile scheme

A **binfile** URL designates a file on your system that contains binary data:

```
put URL "binfile:beachball.gif" into image "Beachball"
```

When you use a **binfile** URL in an expression, LiveCode gets the contents of the file you designate and substitutes it for the URL. The following example puts the contents of a file into a variable:

```
put URL "binfile:picture.png" into pictVar
```

When you put data into a **binfile** URL, LiveCode puts the data into the file:

```
put pictVar into URL "binfile:/Volumes/Backup/pict.png"
put image 1 into "binfile:/image.png"
```

As with local variables, if the file doesn't exist, putting data into it creates the file.

The **binfile** scheme works like the file scheme, except that LiveCode does not attempt to convert end-of-line markers. This is because return and linefeed characters can be present in a binary file but not be intended to mark the end of the line. Changing these characters can corrupt a binary file, so the **binfile** scheme leaves them alone.

The resfile scheme

On Mac OS Classic (and sometimes on OS X systems), files can consist of either a data fork or a resource fork or both.

Important: While LiveCode supports reading and writing resource fork files on Mac OS X, this feature is only intended to help you access and work with legacy files. We do not generally recommend the use of resource forks when designing any new application.

The resource fork contains defined resources such as icons, menu definitions, dialog boxes, fonts, and so forth. A **resfile** URL designates the resource fork of a Mac OS X file:

```
put myBinaryData into URL "resfile:/Disk/Resources"
```

When you use a **resfile** URL in an expression, LiveCode gets the resource fork of the file you designate and substitutes it for the URL.

When you put data into a **resfile** URL, LiveCode puts the data into the file's resource fork.

Note: A **resfile** URL specifies the entire resource fork, not just one resource. To work with individual resources, use the **getResource**, **setResource**, **deleteResource** and **copyResource** functions.

The most common use for this URL scheme is to copy an entire resource fork from one file to another. To modify the data from a **resfile** URL, you need to understand the details of Apple's resource fork format.

Creating a resource fork

Unlike the **file** and **binfile** URL schemes, the **resfile** keyword cannot be used to create a file. If the file doesn't yet exist, you cannot use the **resfile** keyword to create it. To create a new resource file, first use a **file** URL to create the file with an empty data fork, then write the needed data to its resource fork:

```
put empty into URL "file:myFile" -- creates an empty file
put myStoredResources into URL "resfile:myFile"
```

Manipulating URL contents

You use a URL like any other container. You can get the content of a URL or use its content in any expression. You can also put any data into a URL.

http, **ftp**, **binfile**, and **resfile** URLs can hold binary data.

http, **ftp**, and **file** URLs can hold text.

The URL keyword

To specify a URL container, you use the **URL** keyword before the URL, which can use any of the five schemes described above:

```
if URL "http://www.example.net/index.html" is not empty then ...

get URL "binfile:/Applications/Hover.app/data"

put 1+1 into URL "file:output.txt"
```

The **URL** keyword tells LiveCode that you are using the **URL** as a container.

Some properties (such as the **filename** of a player or image) let you specify a URL as the property's value. Be careful not to include the **URL** keyword when specifying such properties, because using the **URL** keyword indicates that you're treating the URL as a container. If you use the **URL** keyword when specifying such a property, the property is set to the contents of the URL, not the URL itself, and this is usually not what's wanted.

Using the content of a URL

As with other containers, you use the content of a URL by using a reference to the URL in an expression. LiveCode substitutes the URL's content for the reference.

If the URL scheme refers to a local file (**file**, **binfile**, or **resfile** URLs), LiveCode reads the content of the file and substitutes it for the URL reference in the expression:

```
answer URL "file:../My File"
-- displays the file's content
put URL "binfile:flowers.jpg" into myVariable
put URL "resfile:Icons" into URL "resfile:New Icons"
```

If the URL scheme refers to a document on another system (**http** or **ftp** URLs), LiveCode downloads the URL automatically, substituting the downloaded data for the URL reference:

```
answer URL "http://www.example.net/files/greeting.txt"
```

Note: If the server sends back an error message--for example, if the file you specify in an **http** URL doesn't exist--then the error message replaces the URL reference in the expression.

Important: When you use an **ftp** or **http** URL in an expression, the handler pauses until LiveCode is finished downloading the URL. If you do not want to block LiveCode when accessing these resources, use the **load URL** form of the command (see below).

Putting data into a URL

As with other containers, you can put data into a URL. The result of doing so depends on whether the URL scheme specifies a file on your system (**file**, **binfile**, or **resfile**) or on another system (**http** or **ftp**).

If the URL scheme refers to a local file (**file**, **binfile**, or **resfile** URLs), LiveCode puts the data into the specified file:

```
put field "My Text" into URL "file:storedtext.txt"
put image 1 into URL "binfile:picture.png"
```

If the URL scheme refers to a document on the Internet (**http** or **ftp** URLs), LiveCode uploads the data to the URL:

```
put myVar into URL "ftp://me:pass@ftp.example.net/file.dat"
```

Because most web servers do not allow HTTP uploads, this usually will not be successful with the **http** scheme.

Chunk expressions and URLs

Like other containers, URLs can be used with chunk expressions to specify a portion of what's in a URL--a line, an item, a word, or a character. In this way, any chunk of a URL is like a container itself. For more information about Chunk Expressions, see the guide on *Processing Text and Data*.

You can use any chunk of a URL in an expression, in the same way you use a whole URL:

```
get line 2 of URL "http://www.example.net/index.html"
put word 8 of URL "file:/Disk/Folder/myfile" into field 4
if char 1 of URL "ftp://ftp.example.org/test.jpg" is "0" then ...
```

You can also specify ranges, and even one chunk inside another:

```
put char 1 to 30 of URL "binfile:/marks.dat" into myVar
answer line 1 to 3 of URL "http://www.example.com/file"
```

Putting data into a chunk

If the URL is local (that is, if it is a **file**, **binfile**, or **resfile** URL), you can put a value into a chunk of the URL:

```
put it into char 7 of URL "binfile:/picture.gif" put return after \
    word 25 of URL "file:../datafile"
put field 3 into line 20 of URL "file:myfile.txt"
```

You can also put a value into a chunk of an **ftp** or **http** URL. Because it's impossible to upload part of a file, LiveCode downloads the file, makes the change, then uploads the file back to the server.

Tip: This method is inefficient if you need to make several changes. In this case, it's faster to first put the URL in a variable, replace the chunk you want to change, then put the variable into the URL:

```
put URL "ftp://me:secret@ftp.example.net/file.txt" into myVar
put field "New Info" after line 7 of myVar
put field "More" into word 22 of line 3 of myVar
put myVar into URL "ftp://me:secret@ftp.example.net/file.txt"
```

This ensures that the file only needs to be downloaded once and re-uploaded once, no matter how many changes you need to make.

URLs and memory

URLs, unlike other containers, are only read into memory when you use the URL in a statement. Other containers – like variables, fields, buttons, and images – are normally kept in memory, so accessing them doesn't increase memory usage.

This means that in order to read a URL or place a value in a chunk of a URL, LiveCode reads the entire file into memory. Because of this, you should be cautious when using a URL to refer to any very large file.

Even when referring to a single chunk in a URL, LiveCode must place the entire URL in memory. An expression such as line 347882 of URL "file:bigfile.txt" may be evaluated very slowly or even not work at all, if insufficient memory is available. If you refer to a chunk of an **ftp** or **http** URL, LiveCode must download the entire file to find the chunk you specify.

If you need to read and write large quantities of data to a file, or seek through the contents of a large file

without loading the entire contents into memory, use the **open file**, **read from file**, **seek** and **close file** commands instead of the URL commands. For more information on these commands see the *LiveCode Dictionary*.

Deleting URLs

You remove a URL with the **delete URL** command.

To delete a local file, you use a **file** or **binfile** URL:

```
delete URL "file:C:/My Programs/test.exe"
delete URL "binfile:../mytext.txt"
```

It doesn't matter whether the file contains binary data or text; for deletion, these URL schemes are equivalent.

Tip: You can also use the **delete file** command to remove a file. To delete the resource fork of a file, you use a **resfile** URL. The following example removes the resource fork along with all resources, but leaves the file in place:

```
delete URL "resfile:/Volumes/Backup/proj.rev"
```

Tip: To delete a single resource instead of the entire resource fork, use the **deleteResource** function.

To remove a file or directory from an FTP server, you use an **ftp** URL:

```
delete URL "ftp://root:secret@ftp.example.org/deleteme.txt"
delete URL "ftp://me:mine@ftp.example.net/trash/"
```

As with creating files, you can use an **http** URL to delete a file, but most HTTP servers are not configured to allow this.

Uploading and Downloading Files

The simplest way to transfer data to an FTP or HTTP server is to use the **put** command to upload, or use the URL in an expression to download.

The Internet library includes additional commands to upload and download files to and from an FTP server. These commands offer more versatile options for monitoring and controlling the progress of the file transfer.

Uploading using the put command

As mentioned above, putting something into an **ftp** or **http** URL uploads the data to the server:

```
put myVariable into URL
"ftp://user:pass@ftp.example.org/newfile.txt"
```

If you use the **put** command with a **file** or **binfile** URL as the source, the file is uploaded:

```
put URL "file:newfile.txt" into URL
"ftp://user:pass@ftp.example.org/newfile.txt"
```

When you upload data in this way, the operation is blocking: that is, the handler pauses until the upload is finished. (See below for details on how to create a file transfer that is not blocking.) If there is an error, the error is placed in the **result** function:

```
put field "Data" into URL myFTPDestination
if the result is not empty then beep 2
```

Important: Uploading or downloading a URL does not prevent other messages from being sent during the file transfer: the current handler is blocked, but other handlers are not. For example, the user might click a button that uploads or downloads another URL while the first URL is still being uploaded. In this case, the second file transfer is not performed and the **result** is set to "Error Previous request has not completed." To avoid this problem, you can set a flag while a URL is being uploaded, and check that flag when trying to upload or download URLs to make sure that there is not already a file transfer in progress.

Downloading using a URL

Referring to an **ftp** or **http** URL in an expression downloads the document.

```
put URL "ftp://ftp.example.net/myfile.jpg" into image 1
get URL "http://www.example.com/newstuff/newfile.html"
```

If you use the **put** command with a **file** or **binfile** URL as the destination, the document is downloaded to the file:

```
put URL "ftp://ftp.example.net/myfile.jpg" into URL \
"binfile:/Disk/Folder/myfile.jpg"
```

Non-blocking transfers

When you transfer a file using URL containers, the file transfer stops the current handler until the transfer is done. This kind of operation is called a blocking operation, since it blocks the current handler as long as it's going on.

If you want to transfer data using *http* without blocking, use the **load** command. If you want to transfer large files using *ftp*, use the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile** commands.

Non-blocking file transfers have several advantages:

Since contacting a server may take some time due to network lag, the pause involved in a blocking operation may be long enough to be noticeable to the user.

If a blocking operation involving a URL is going on, no other blocking operation can start until the previous

one is finished. If a non-blocking file transfer is going on, however, you can start other non-blocking file transfers. This means that if you use the library commands, the user can begin multiple file transfers without errors.

During a non-blocking file transfer, you can check and display the status of the transfer. This lets you display the transfer's progress and allow the user to cancel the file transfer.

Using the load command

The **load** command downloads the specified document in the background and places it in a cache. Once a document has been cached, it can be accessed nearly instantaneously when you use its URL, because LiveCode uses the cached copy in memory instead of downloading the URL again.

To use a file that has been downloaded by the load command, refer to it using the URL keyword as usual. When you request the original URL, LiveCode uses the cached file automatically.

For best performance, use the **load** command at a time when response speed isn't critical (such as when your application is starting up), and only use it for documents that must be displayed quickly, such as images from the web that will be shown when you go to the next card.

Checking status when using the load command

While a file is being transferred using the load commands, you can check the status of the transfer using the **URLStatus** function. This function returns the current status of a URL that's being downloaded or uploaded:

```
local tUrl
put "ftp://ftp.example.com/myfile.txt" into tUrl
put the URLStatus of tUrl into field "Current Status"
```

The **URLStatus** function returns one of the following values:

- *queued* : on hold until a previous request to the same site is completed
- *contacted* : the site has been contacted but no data has been sent or received yet
- *requested* : the URL has been requested
- *loading bytesTotal, bytesReceived* : the URL data is being received
- *uploading bytesTotal, bytesReceived* : the file is being uploaded to the URL
- *cached* : the URL is in the cache and the download is complete
- *uploaded* : the application has finished uploading the file to the URL
- *error* : an error occurred and the URL was not transferred
- *timeout* : the application timed out when attempting to transfer the URL

To monitor the progress of a file transfer or display a progress bar, you check the **URLStatus** function repeatedly during the transfer. The easiest way to do this is with timer based messaging – see the section of the same name in the *LiveCode Script* guide, for more information.

Canceling a file transfer & emptying the cache

To cancel a transfer initiated with the load command and empty the cache, use the **unload** command.

```
unload URL "http://example.org/new\_beta"
```

Uploading and downloading large files using FTP

The Internet library provides a number of commands for transferring larger files via FTP without blocking.

- **libURLftpUpload** uploads data to an FTP server
- **libURLftpUploadFile** uploads a file to an FTP server
- **libURLDownloadToFile** downloads a file from an FTP server to a local file

The basic effect of these commands is the same as the effect of using URLs: that is, the data is transferred to or from the server. However, there are several differences in how the actual file transfer is handled. Because of these differences, the library commands are more suitable for uploads and downloads, particularly if the file being transferred is large.

The following sets of statements each show one of the Internet library commands, with the equivalent use of a URL:

```
libURLftpUpload myVar,"ftp://me:pass@example.net/file.txt"
put myVar into URL "ftp://me:pass@example.net/file.txt"

libURLftpUploadFile "test.data","ftp://ftp.example.org/test"
put URL "binfile:test.data" into URL "ftp://ftp.example.org/test"

libURLDownloadToFile "ftp://example.org/new\_beta","/HD/File"
put URL "ftp://example.org/new\_beta" into URL "binfile:/HD/File"
```

Using callback messages

When you start a file transfer using the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile** command, you can optionally specify a callback message, which is usually a custom message that you write a handler for. This message is sent whenever the file transfer's **URLStatus** changes, so you can handle the callback message to handle errors or to display the file transfer's status to the user.

The following simple example demonstrates how to display a status message to the user. The following handlers might be found in a button's script:

```
on mouseUp
    local tUrl
    put "ftp://example.org/new\_beta" into tUrl
    libURLDownloadToFile tUrl,"/HD/Latest Beta","showStatus"
end mouseUp

on showStatus theURL
    put the URLStatus of theURL into field "Status"
end showStatus
```

When you click the button, the **mouseUp** handler is executed. The **libURLDownloadToFile** command begins the file transfer, and its last parameter specifies that a *showStatus* message will be sent to the button whenever the **URLStatus** changes.

As the **URLStatus** changes periodically throughout the download process, the button's *showStatus* handler is executed repeatedly. Each time a *showStatus* message is sent, the handler places the new status in a field. The user can check this field at any time during the file transfer to see whether the download has started, how much of the file has been transferred, and whether there has been an error.

If a file transfer was started using the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile** command, you can cancel the transfer using the **unload** command.

Uploading, downloading, and memory

When you use a URL as a container, LiveCode places the entire URL in memory. For example, if you download a file from an FTP server using the **put** command, LiveCode downloads the whole contents of the file into memory before putting it into the destination container. If the file is too large to fit into available memory, a file transfer using this method will fail (and may cause other unexpected results).

The library commands **libURLftpUpload**, **libURLftpUploadFile**, and **libURLDownloadToFile**, however, do not require the entire file to be loaded into memory. Instead, they transfer the file one piece at a time. If a file is (or might be) too large to comfortably fit into available memory, you should always use the library commands to transfer it.

Using a stack on a server

Ordinarily, you use stack files that are located on a local disk. You can also open and use a stack that is located on an FTP or HTTP server. Using this capability, you can update an application by downloading new stacks, make new functionality available via the Internet, and even keep most of your application on a server instead of storing it locally.

Going to a stack on a server:

As with local stack files, you use the **go** command to open a stack that's stored on a server:

```
go stack URL "http://www.example.org/myapp/main.rev"
go stack URL "ftp://user:pass@example.net/secret.rev"
```

Note: For such a statement to work, the stack file must have been uploaded as binary data, uncompressed, and not use encodings such as BinHex.

Tip: If you need to download a large stack, use the **load** command to complete the download before using the **go** command to display the stack. This allows you to display a progress bar during the download.

LiveCode automatically downloads the stack file. The main stack of the stack file then opens in a window, just as though you had used the **go** command to open a local stack file.

You can go directly to a specific card in the stack:

```
local tStackUrl
put "http://www.example.org/myapp/main.rev" into tStackUrl
go card "My Card" of stack URL tStackUrl
```

To open a substack instead, use the substack's name:

```
local tStackUrl
put "http://www.example.org/myapp/main.rev" into tStackUrl
go stack "My Substack" of stack URL tStackUrl
```

Using a compressed stack

You cannot directly open a stack that's compressed. However, since the stack URL is a container, you can use the URL as the parameter for the **decompress** function. The function takes the stack file data and decompresses it, producing the data of the original stack file. You can open the output of the function directly as a stack.

The following statement opens a compressed stack file on a server:

```
go decompress(stack URL "http://www.example.net/comp.gz")
```

The statement automatically downloads the file "comp.gz", uncompresses it, and opens the main stack of the file.

Saving stacks from a server

When a stack is downloaded using the **go** command, it's loaded into memory, but not saved on a local disk. Such a stack behaves like a new (unsaved) stack until you use the **save** command to save it as a stack file.

Note: Saving a stack that has been downloaded with the **go** command does not re-upload it to its server. To upload a changed stack, you must save it to a local file, then use one of the methods described in this topic to upload the file to the server.

Other Internet Commands

The Internet library has a number of additional commands for working with web forms, ftp commands, custom settings and troubleshooting. These commands are documented in more detail the LiveCode Dictionary.

Launching the User's Browser with a URL

To launch the default browser with a URL, use the **launch URL** command.

```
launch URL "http://www.livecode.com/"
```

Note: To render web pages within LiveCode, instead of launching an external browser, use the **revBrowser**. See the section on **revBrowser** for more information.

Working with Web Forms

To post data to a web form, use the **post** command. To encode data to make it suitable for posting, use the **libUrlFormData** function. To create multi-part form data (as described in RFC 1867) use the **libUrlMultipartFormData** function. To add data to a multipart form one part at a time, use the **libUrlMultipartFormAddPart** function. This can be useful if you need to specify the mime type or transfer encoding for each part.

Working with FTP

For details on basic uploading and downloading using FTP, see the section above.

The following commands provide additional capabilities when working with the ftp protocol:

- **libURLSetFTPStopTime** : Sets the timeout value for FTP transfers.
- **libURLSetFTPMode** : Switches between active and passive mode for FTP transfers.
- **libURLSetFTPListCommand** : Switches between sending LIST or NLST formats when listing the contents of an FTP directory.
- **libURLftpCommand** : sends an ftp command to an ftp server.
- **libURLftpUpload** : uploads data. See the section above for more details.
- **libURLftpUploadFile** : uploads a file, without loading the entire file into memory. See the section above for more details.
- **libURLDownloadToFile** – downloads data to a file, without loading the entire data into memory. See the section above for more details.

HTTP methods and http URLs

The basic operations used by the HTTP protocol are called methods. For **http** URLs, the following HTTP methods are used under the following circumstances:

- GET: when an **http** URL in an expression is evaluated
- PUT: when you put a value into an **http** URL
- POST: when you use the **post** command
- DELETE: when you use the **delete URL** command with an **http** URL

Note: Many HTTP servers do not implement the PUT and DELETE methods, which means that you can't put values into an **http** URL or delete an **http** URL on such servers. It's common to use the FTP protocol instead to upload and delete files; check with your server's administrator to find out what methods are supported.

HTTP headers

When LiveCode issues a GET or POST request, it constructs a minimal set of HTTP headers. For example, when issued on a Mac OS system, the statement:

```
put URL "http://www.example.org/myfile" into myVariable
```

results in sending a GET request to the server:

```
GET /myfile HTTP/1.1 Host: 127.0.0.0 User-Agent: LiveCode (MacOS)
```

You can add headers, or replace the Host or User-Agent header, by setting the **HTTPHeaders** property before using the URL:

```
set the HTTPHeaders to "User-Agent: MyApp" & return \
  & "Connection: close"
put URL "http://www.example.org/myfile" into myVariable
```

Now the request sent to the server looks like this:

```
GET /myfile HTTP/1.1 Host: 127.0.0.0 User-Agent: MyApp Connection: close
```

The **ftp** URL scheme can be used to create a new file to an FTP server. As with the **file** and **binfile** schemes, putting something into the URL creates the file:

```
put dataToUpload into URL
"ftp://jane:pass@ftp.example.com/newfile.dat"
```

Tip: You can create an FTP directory by uploading a file to the new (nonexistent) directory. The directory is automatically created. You can then delete the file, if you wish, leaving a new, empty directory on the server:

```
-- Create an empty file in the nonexistent directory:
put empty into URL "ftp://jane:pass@example.com/newdir/dummy"

-- Delete unwanted empty file to leave new directory:
delete URL "ftp://jane:pass@example.com/newdir/dummy"
```

Additional Transmission Settings

The following commands provide additional customization options for the Internet library:

- **libUrlSetExpect100** : Allows you to set a limit to the size of data being posted before requesting a continue response from the server.
- **libURLSetCustomHTTPHeaders** : Sets the headers to be sent with each request to an HTTP server. See also the section on HTTPHeaders above.
- **libURLFollowHttpRedirects** : Specify that GET requests should follow HTTP redirects and GET the page redirected to.
- **libUrlSetAuthCallback** : Sets a callback for handling authentication with http servers and proxies.

Troubleshooting

The following commands and functions can be useful when debugging an application that uses the Internet library.

- **resetAll** : Closes all open sockets and halts all pending Internet operations.

Caution: The **resetAll** command closes all open sockets, which includes any other sockets opened by your application and any sockets in use for other uploads and downloads. Because of this, you should avoid routine use of the **resetAll** command. Consider using it only during development, to clear up connection problems during debugging.

- **libURLErrorData** : Returns any error that was caused during a download that was started with the load command.
- **libURLVersion** : Returns the version of the Internet library.
- **libURLSetLogField** : Specifies a field for logging information about uploads and downloads on screen.
- **libURLLastRHHeaders** : Returns the headers sent by the remote host in the most recent HTTP transaction.
- **libURLLastHTTPHeaders** : Returns the value of the httpHeadersproperty used for the previous HTTP request.

revBrowser – Rendering a Web Page within a Stack

Use the revBrowser commands to render a web page within a stack. RevBrowser uses WebKit (Safari) on Mac OS X and Internet Explorer on Windows. Currently RevBrowser is not supported under Linux.

To create a browser object in a stack, use the **revBrowserOpen** function. This function takes the **windowID** for the stack you want to open the browser in and a URL. Please note that the **windowID** is not the same as the stack's ID property.

```
put the windowid of this stack into tWinID
put revBrowserOpen(tWinID,"http://www.google.com") into sBrowserId
```

To set properties on the browser, use the **revBrowserSet** command. The following commands makes the border visible then sets the rectangle to be the same as an image named "browserimage":

```
revBrowserSet sBrowserId, "showborder","true"
revBrowserSet sBrowserId, "rect",rect of img "browserimage"
```

To close a browser when you finished with it, use the **revBrowserClose** command. This command takes the **windowID** for the stack containing the browser:

```
revBrowserClose sBrowserId
```

RevBrowser supports a number of settings and messages. You can intercept a message whenever the user navigates to a link, prevent navigation, intercept clicks in the browser, requests to download files or to open a new window.

For a complete list of commands that operate on RevBrowser, open the *LiveCode Dictionary* and type "browser" into the filter box.

SSL and Encryption

LiveCode includes support for using Secure Sockets Layer and the https protocol. It also includes an industrial strength encryption library you can use to encrypt files or data transmissions.

Encrypting and Decrypting Data

To encrypt data, use the **encrypt** command. The **encrypt** command supports a wide variety of industry standard methods of encryption. The list of installed methods can be retrieved by using the **cipherNames** function. To decrypt data, use the **decrypt** command. For more information on these features, see the *LiveCode Dictionary*.

Tip: If you are using the encryption library on a Windows system, it is possible that another application will have installed DLLs that use the same name as the ones included with LiveCode to support encryption. You can force your application to load LiveCode's SSL DLLs by setting the \$PATH environment variable before loading the library.

```
put $PATH into tOldPath
put <path to SSL DLLs> into $PATH
get the cipherNames -- Force loading of the SSL DLLs
put tOldPath into $PATH
```

Connecting using HTTPS

You may connect and download data from a URL using **https** in the same way that you access an http URL.

```
put URL "https://www.example.com/store.php"
```

If there is an error, it will be placed into `the result`. If you need to include a user name and password you can do so in the following form:

```
https://user:password@www.example.com/
```

Implementing your own secure protocols

To implement your own secure protocol, use the **open secure socket** variant of the **open socket** command. You can specify whether or not to include certification, a certificate and a key. For more information on the **open socket** command, see the *LiveCode Dictionary*.

Writing your own protocol with sockets

If you need to implement your own protocol, you can do so using LiveCode's socket support. To understand this chapter it is assumed you understand the basics of how the Internet works, including the concepts of sockets, IP addresses and ports. More information on these concepts can be found in Wikipedia.

Tip: The standard protocols that LiveCode support such as http and ftp, discussed earlier in this chapter, have all been implemented as a scripted library with LiveCode's socket support. You can examine this library by running `edit script of stack "revlibURL"` in the Message Box. Beware, this library is not for the faint of heart. If you change anything, LiveCode's Internet commands may cease to operate.

Opening a connection

To open a connection use the **open socket** command. The following command opens a connection to the IP address specified in the `tIPAddress` variable and the port specified in the `tPort` variable. It specifies that LiveCode should send the message `chatConnected` when a connection has been established.

```
open socket (tIPAddress & ":" & tPort) with message "chatConnected"
```

To open a secure socket, use the **open secure socket** variant of the command. To open a UDP

datagram socket, use the **open datagram socket** variant of the command. For more information on these variants, see the *LiveCode Dictionary*.

Looking up a host name or IP address

You may look up an IP address from a host name with the **hostNameToAddress** function. For example, to get the IP address for the livecode.com server:

```
put hostNameToAddress("www.livecode.com") into tIPAddress
```

To get the host name of the local machine, use the **hostName** function. To look up the name from an IP address, use the **hostAddressToName** function.

Reading and writing data

Once LiveCode opens a connection, it will send a `chatConnected` message. To receive data, use the **read from socket** command. The following message reads data from the socket and sends a `chatReceived` message when reading is completed.

```
on chatConnected pSocket
  read from socket pSocket with message chatReceived
end chatConnected
```

Once reading from the socket is completed the `chatReceived` message can be used to process or display the data. It can then specify that it should continue to read from the socket until more data is received, sending another `chatReceived` message when done.

```
on chatReceived pSocket, pData
  put pData after field "chat output"
  read from socket pSocket with message "chatReceived"
end chatReceived
```

To write data to the socket, use the **write** command:

```
write field "chat text" to socket tSocket
```

Disconnecting

To disconnect, use the **close socket** command. You should store a variable with details of any open sockets and close them when you have finished using them or when your stack closes.

```
close socket (tIDAddress & ":" & tPort)
```

Listening for and accepting incoming connections

To accept incoming connections on a given port, use the **accept connections** command. The following example tells LiveCode to listen for connections on port 1987 and send the message `chatConnected` if a connection is established. You can then start to read data from the socket in the `chatConnected` handler.

```
accept connections on port 1987 with message chatConnected
```

Handling errors

If there is an error, LiveCode will send a **socketError** message with the address of the socket and the error message. If a socket is closed a **socketClosed** message will be sent. If a socket times out waiting for data a **socketTimeout** message will be sent. To get a list of sockets that are open, use the **openSockets** function. You can set the default timeout interval by setting the **socketTimeOutInterval** property. For more details on all of these features, see the *LiveCode Dictionary*.

Tip: You can see a complete implementation of a basic client server "chat" application by navigating to Documentation -> Getting Started -> Sample Projects -> Internet Chat - creating a custom protocol using sockets -> Launch. Most of the scripts for the "server" stack are in the "start server" button. Most of the scripts for the client are in the stack script for the "chat client" stack.

Working With Databases

Introduction

With the LiveCode Database library, your application can communicate with external SQL databases. You can get data from single-user and multi-user databases, update data in them, get information about the database structure, and display data from the database in your stack. For a discussion of when it is appropriate to use an external database with LiveCode, see the topic *When to Use a Database* in the *LiveCode Script* guide.

This guide discusses how to install necessary software to communicate with databases, and how to use the Database library to communicate between LiveCode and a database.

This topic does not include discussion of how to set up and create a SQL database, which is beyond the scope of the LiveCode documentation.

To fully understand this topic, you should know how to write short scripts and should understand the basic concepts of SQL databases (rows and columns, database cursors, and SQL queries).

A few terms used in this topic, such as "field" and "cursor", are part of the standard terminology for working with databases, but have a different meaning in the context of LiveCode development. When going back and forth between database work and more general application development, be sure you understand which meaning is applicable in the context you're currently working in. When referring to database-specific terms, the documentation usually uses phrases like "database field" or "database cursor" to remind you of the context. See the Glossary within the product documentation for definitions of any term you are unsure of.

Introduction to Database Access

A database is an external resource that holds information, structured in a special form for quick access and retrieval. Databases can be:

- any size from small to extremely large
- located on the same system as the application or on a remote server
- accessible by one user at a time or by many users at once

SQL Databases

A SQL database is a database that you access and control using SQL, a standard database-access language which is widely supported. You use SQL queries (statements in the SQL language) to specify the part of the database you want to work with, to get data, or to make changes to the database.

LiveCode's database access is fully-featured. You can send any SQL statement to a database. You can open multiple databases (or multiple connections to the same database), maintain multiple record sets (database cursors) per connection, and send and receive binary data as well as text. You can do all this using the commands and functions in the *Database library*.

To see a list of LiveCode terms in the Database library, open the *Dictionary*, and type "database" into the search filter field.

The Basics of Database Structure

A database is built of records, which in turn are built out of database fields. A field is the smallest part of a database that can be separately addressed. Each database field contains a particular kind of

information. This might be a name, a file path, a picture, or any other kind of information. Each record contains one value for each of its fields. A set of records is called a database table, and one or more tables comprise a database.

Here's an example: suppose you have a database of customers for your business. The fields of this database might include the customer name, a unique customer ID number, and shipping address. Each record consists of the information for a single customer, so each record has a different customer name, shipping address, and so on.

Note: You may have noticed that the database structure being described resembles a multiple-card stack that has the same fields on each card. A database field is like a field in a stack, and a record is like a card. A stack set up this way can act as a database, in fact, but lacks some of the features of an external database, such as the ability to perform SQL queries and the ability to perform robustly when accessed by more than one user.

You can also think of the set of customer records as a grid (like a spreadsheet). Each row is a record, and each column is a field, so each cell in the grid contains a different piece of information about a particular customer. Here's an example:

ID	Customer Name	Address	Country
123	Jane Jones	234 E. Street	U.K.
836	Acme Corporation	PO Box 23788	USA
823	CanCo, Inc.	1 CanCo Blvd.	Japan

Figure 59 – Example Database Grid

There are three rows in this grid (each is the record for a particular customer) and four columns (each is one of the fields).

A row of the database means one single customer record, which has one value for each field. A column of the database means the set of all values for one of the fields, one for each record (for example, the set of all customer addresses).

More generally, each row describes one of the things in the database, and each column describes a particular state for each thing in the database.

The set of all customer records makes a table. Your database might include only this table, or it might include other related tables, such as a list of all sales. You can also connect related data from different tables of the same database. For example, if each record in the Sales table includes the ID number of the customer who bought the product, you can link all the sales records for a customer to that customer's record in the Customers table.

SQL and Record Sets – Database Cursors

SQL works primarily with sets of records, rather than individual rows. When you send a SQL query to a database, the query typically selects certain records which you can perform further operations on. The set of records resulting from a SQL query is called a database cursor, or record set. SQL queries let you describe the characteristics of the records you require, instead of processing each record one by one to find out whether it matches your criteria.

For example, consider the customer table we talked about in the previous section. To work with only US customers, you can write a SQL query that selects only records where the country field is "USA". This subset of records then becomes a record set. You can find out more about the fields and records contained in this record set, and move from record to record within this subset of the database. You can create more than one record set to work with more than one set of records at a time.

Note: Different database implementations have different limitations on movement within a record set. For example, some databases won't let you move backward within a record set: instead, you must start at the first record and move forward in order to examine the data.

Choosing a Database

LiveCode directly supports the following database implementations:

- MySQL
- SQLite
- PostgreSQL

LiveCode also supports connecting to a database via ODBC. You can use ODBC to use Access, FileMaker, MS SQL Server and many other database implementations. See below for more information about ODBC.

LiveCode's database commands and functions use the same syntax regardless of what type of database you are connecting to. You don't need to learn a separate database language for each type. Instead, when you first open a database with the **revOpenDatabase** function, you specify the type as one of the parameters so LiveCode knows what type of database it's dealing with. The Database library handles the details of each type behind the scenes for you.

Reasons to Choose a Database Type

Which type of database to choose depends on a number of factors. If you need to work with an existing database, or already have a database manager installed, the decision is made for you. Likewise, if you're already an expert at a particular database implementation, you'll probably prefer to go on using that one.

Other factors in your choice may include price, performance, licensing model (commercial or open source), and platform support. If your users are on a particular set of platforms, you'll need to choose a database that is supported on all those platforms. Runtime do not endorse any particular database type, the information provided on the different types is for informational purposes only.

Overview of ODBC

Open Database Connectivity (ODBC) is a system that allows developers to access any type of compatible database in a standard way.

To communicate with a database, you usually have to add code that uses the database's own proprietary protocols. Without ODBC, in order to create a program that can communicate with--for example--FileMaker, Access, and Oracle databases, LiveCode would have to include code for three different database protocols. With ODBC, LiveCode can communicate with any of these database types using the same code.

ODBC Managers

To work with databases through ODBC, you need two pieces of software: an ODBC manager, plus a database driver for the specific database type you're using.

Windows and OS X include ODBC software with the operating system. For Linux systems, you can download an ODBC manager and a set of drivers (see the section below titled "Software for Database Access" for more information).

Performance for Direct Access Versus ODBC Access

Typically, accessing a database via ODBC takes more configuration and is slower than accessing the database directly. For this reason, LiveCode provides the ability to access MySQL, PostgreSQL databases directly without going through the ODBC protocol. This ability will be valuable for anyone doing complex or extensive professional database work.

The syntax of the functions in the Database library is identical for all database types, so you do not need to rewrite your scripts to take advantage of the increased efficiency of direct access.

Software for Database Access

To provide connectivity to databases, LiveCode works with database drivers--software that translates application requests into the protocol required by a specific database.

Finding Database Drivers

Database drivers for certain database types are included in the LiveCode distribution. (The list of included database types depends on the platform.) The sections below list which database drivers are included with which platforms.

If you have installed LiveCode, you have all the software needed to use the included database types. For other database types, you will need to obtain the appropriate database drivers before you can work with those databases.

Important: This section includes links to third-party web sites and contains information about third-party software. This information is provided for your convenience, but LiveCode is not responsible for the software packages and sites referenced. Runtime regrets that it cannot provide any support for installation and configuration of any databases.

MySQL

MySQL database drivers are included as part of the LiveCode installation on Linux, Mac OS X, and Windows systems.

PostgreSQL

A PostgreSQL database driver is included as part of the LiveCode installation on Linux, Mac OS X and Windows systems.

SQLite

Drivers for accessing this database are included with LiveCode. No additional installation is necessary.

ODBC managers and database drivers

To use a database via ODBC, you must install the necessary ODBC software for your platform. (Some operating systems include an ODBC installation.) ODBC software includes one or more database drivers, plus an ODBC manager utility.

ODBC on Windows systems

Windows systems include the MDAC (Microsoft Data Access Components) package as part of the standard system installation. To configure ODBC on Windows systems, use the ODBC Data Sources control panel.

ODBC on Mac OS X systems

OS X includes iODBC software as part of the standard system installation. To configure ODBC on OS X systems, use the ODBC Administrator application in the Utilities folder.

ODBC on Linux systems

LiveCode supports iODBC and UnixODBC on Linux systems. You can download the iODBC software from

the iODBC web site at <http://www.iodbc.org/>. You can download the unixODBC software from the unixODBC web site at <http://www.unixodbc.org>.

Creating a DSN for ODBC access

Once you have installed the necessary software, you use the ODBC manager to create a DSN, which is a specification that identifies a particular database.

You use the DSN to connect to the database via ODBC. The following example opens a connection to a database whose DSN is named "myDB":

```
get revOpenDatabase("ODBC", "myDB", , "jones", "pass")
```

One of the advantages of setting up a DSN is that if you wish to change the location of the database, you only have to edit the DSN settings, not your application code. You can think of a DSN as a kind of shortcut or alias to your database.

Working with Media

Introduction

One of the most popular uses for LiveCode is to create full blown multimedia applications. Even if you aren't creating a traditional multimedia application, many applications require a compelling user interface.

This chapter details LiveCode's media support. We discuss the image features and capabilities: how to import and export images in a variety of formats, how to manipulate images within LiveCode, working with masks, the clipboard and screen capture, right through to animated GIFs. We detail the vector graphic features and explain how to manipulate vector graphics by script. We then cover the video and audio feature set. Then we show you how to create custom themed buttons. Finally we give an overview of the support for visual transition effects.

Bitmap Images

LiveCode supports a wide variety of image formats, including the popular PNG and JPEG formats. PNG images are space efficient and have full support for alpha channels. JPEG images are good for displaying photos. For more details see the table below.

Format	Export	Mask	Comments
PNG	Yes	Yes, 1-bit or alpha channel	Supports gamma adjustment, supports alpha channels, supports interlacing
JPEG	Yes	No	Supports progressive JPEGs; lossy compression. Export allows you to set the level of compression
GIF	Yes	1-bit	GIF87a and GIF89a; supports animation; supports interlaced GIFs; maximum 256 colors
BMP	No	No	Uncompressed
PBM	Yes	No	1-bit (black and white)
PGM	No	No	Grayscale
PPM	No	No	
XBM	No	No	
XPM	No	No	1-bit (black and white)
XWD	No	No	
PICT	No	No	uncompressed

As you can see from the table above, a number of the supported image formats can also be exported by LiveCode.

You can modify images using LiveCode's paint tools, or manipulate the binary data by script or using an external.

You can create images from any of LiveCode's native objects, including buttons, fields and graphics. These can then be exported in a number of formats. You can copy images from the clipboard or export images to the clipboard.

LiveCode can capture a portion of the screen, or the entire screen.

Importing Images

To import an image, choose File -> Import As Control -> Image File. Select the image file you want to

import. This will import the image file into a new image object on the current card. This is equivalent to executing the **import paint** command in the Message Box.

Note: If you want to reuse an image throughout your application, for example as part of a custom skin for your application, create a substack and import all your images into that. You can then reference them throughout your stack file. For more information, see the section *Creating Custom Skins*, below.

Importing Referenced Images

To reference an image file on disk choose File -> New Referenced Control -> Image File. This creates an image object on the current card and sets its **fileName** property to the path of the image file you selected. Referenced images do not expand the size of your stack and are only loaded from disk into memory when you navigate to the current card or otherwise display them on screen. Referenced images are ideal where you want to update the image in an image editor and see the changes in LiveCode without having to re-import.

You may wish to consider creating a folder next to your stack file to contain image files then using the Inspector to modify the file path to the image to be a referenced path. This allows you to move the stack and folder of images together onto a different system. for more information on referenced file paths, see the section *File Name Specifications and Paths*.

You can use the standalone builder to copy referenced images into a directory (updating each image's **fileName** property) or to copy referenced images into your stack. For more information see the chapter *Deploying your Application*.

Important: You cannot use the paint tools or manipulate the binary data of referenced images. You must import them first. If you want to modify the original file, you can make changes then export the image – see below for more details.

Import using Screen Capture

To import an image by capturing a portion of the screen, choose File -> Import As Control -> Snapshot. Select the region of the screen you want to import.

To take a screen capture by script, use the **import snapshot** command. To specify a section of the screen to import from without displaying the crosshairs use **import snapshot from rect**:

```
import snapshot from 100,100,200,200
```

This will create an image object on the current card from the rectangular area specified.

Creating Images

To create an image, drag an image object from the Tools palette to your stack. You may now paint on the image using the paint tools, set the **fileName** reference to an image or manipulate the binary data of the image.

Using the Paint Tools

To access the paint tools, press the fold out triangle at the bottom right of the Tools palette.

Tool	Usage	Keyboard Modifiers
□ Select	Drag to select a rectangular area of an image	Shift constraints to a square; command / control duplicates selection
□ Bucket	Fills shapes with color. Will fill any pixel connected to the pixel you click with the brush color	Control-click to fill with transparency
□ Spray can	Draw an airbrushed color using the brush shape	Control-click to spray with transparency
□ Eraser	Remove color from an area leaving it transparent. Uses the brush shape	
□ Polygon	Draw polygon shape (hold down on the rectangle shape to select this shape)	Shift constrains lines angles to multiples of 22.5°; control creates transparency
□ Line	Draw a straight line	Shift constrains lines angles to multiples of 22.5°; control creates transparency
□ Freehand	Draw a freehand curve (hold down on the line shape to select this shape). If the filled option is chosen the beginning and end of the curve are joined automatically when you finish drawing	Alt / option prevents drawing line border; control creates transparency
□ Pencil	Draw a single-pixel-width freehand line	Control creates transparency
□ Brush	Draw brush strokes using the brush shape	Control creates transparency; command click to magnify
Fill		
□ (brush) color	Select a color to fill shapes or use with the brush tools	
□ Line color	Select a color to draw lines or use with the pencil tool	
□ Brush shape	Choose a brush shape for use with the brush, eraser and airbrush tools	

To magnify an image, right click it with the pointer tool and choose Magnify from the menu.

When you edit an image, it will be recompressed into the format specified by the **paintCompression** global property.

Scripting with the Paint Tools

Painting by script can be useful if you want the user to be able to see each paint action. If you want to manipulate the data of an image more efficiently off screen, see the next section.

To control the paint tools by script, create an image then choose the paint tool you want to use. Set the appropriate brush, pattern or line size then use the drag command to paint.

The following example creates an image, chooses the brush tool, selects a small circular brush shape, selects a red color, then draws a line:

```
-- set the size of the image
set the rect of the templateImage to 100,100,400,400
create image
choose brush tool
set the brush to 8
set the brushColor to red -- could use an RGB triplet here
set the dragSpeed to 20 -- very slow
drag from 120,120 to 300,300
```

For more information, see the entries for the **templateImage**, **tool**, **brush**, **brushColor**, **brushPattern**, **dragSpeed**, **penColor** and **penPattern** in the *LiveCode Dictionary*.

You may reduce the size of an image using the **crop** command.

You may rotate an image using the **rotate** command.

To adjust the quality of the scaling algorithm used when scaling an image, set the **resizeQuality** property before setting the image's rect.

Manipulating Binary Image Data

To manipulate the binary data of an image, use the image's **imageData** property. This property returns the color and transparency value of each pixel in the image in a consistent format regardless of the format the image is saved in. The **imageData** is stored as binary, with each pixel represented by 4 bytes. To convert it to and from RGB values use the **byteToNum** and **numToByte** functions.

For example, the numeric value of the red, green and blue channels respectively for the tenth pixel are given by the expressions:

```
charToNum(char (4 * 9 + 2 of the imageData of image <image>))
charToNum(char (4 * 9 + 3 of the imageData of image <image>))
charToNum(char (4 * 9 + 4 of the imageData of image <image>))
```

To manipulate the binary data of an image using an external, use the **imagePixMapID** property.

When you set the **imageData** of an image the image will be recompressed into the format specified by the **paintCompression** global property.

Rendering an Image from Objects

Use the **import snapshot** command to create an image from objects or a region of a stack. Instead of specifying a rectangle in global coordinates (as described above) specify a stack or object.

Note: Unlike screen capturing, the stack or object you specify to import an image from does not need to be displayed on screen. You can create a layout off screen in an invisible stack then render it into an image.

To import a snapshot of a region of a stack:

```
import snapshot from 100,100,200,200 of stack "Layout"
```

To import a snapshot of an object:

```
import snapshot from button 5 of stack "objects"
```

The import snapshot command creates a new image in the current **defaultStack**. The image is encoded using the current **paintCompression** format.

To save this snapshot directly to a file instead of creating an image, use the export snapshot command:

```
export snapshot from the selectedObject to file "snap.jpg" as JPEG
```

Exporting Images

To export an image in the current format that it is stored in, put it into a binary file using the URL commands. The following example prompts the user to select a file then export the image into it:

```
ask file "Select a file:"
put image "picture" into URL ("binfile:" & it)
```

To export an image in a different format, use the **export** command.

```
export image "picture" to file "picture.png" as PNG
```

You may also export an image to a variable. See the **export** command in the *LiveCode Dictionary* for more information.

Copying and Pasting Images

To copy an image internally without using the system clipboard, simply put it into a variable or into another image.

```
put image 1 into image 2
```

To recompress the image in a different format, use the **export** command to export it to a variable then put that variable into an image.

To copy an image to the clipboard, use the **copy** command.

```
copy image 1
```

To paste an image from the clipboard, use the **paste** command.

To transfer the binary data of an image to and from the clipboard get and set the **clipboardData["image"]** property. See the **clipboardData** entry in the *LiveCode Dictionary* for more information.

Working with Animated GIFs

You can import an animated GIF image in the same way you import other images.

Set the **repeatCount** property to specify how often to play the animation. Setting the **repeatCount** to 0 pauses the animation and setting it to -1 causes it to repeat forever.

To change the current frame set the **currentFrame** property.

Note: If you use an animated GIF as a button icon, it will play simultaneously in each button it is used for.

Working with Vector Graphics

In addition to bitmap graphics, LiveCode also supports vector graphics. You can create vector graphics using the graphics tools, or by script. You can manipulate them interactively in the IDE or by script. You can relayer them, export a description of them or convert them to bitmap format.

Tip: Animation Engine is a 3rd party library that includes a set of functions for programming interactive graphic animations. See the Related Software section of our web site for more information.

The Vector Graphic Tools

To see the graphic tools, unfold the triangle at the bottom right of the tools palette. The graphics tools are located above the paint tools on the tools palette. The graphic tools operate in the same way as the paint tools, except that each time you draw a shape a new graphic object is created. Unlike paint graphics, you can resize and adjust graphic objects after you have created them. For more information on the individual shapes, see the section on *Paint Tools*, above.

Creating Graphics by Script

To create graphics by script, set properties of the **templateGraphic** then use the **create graphic** command. For more information on template objects, see the section *Creating objects off screen using the template objects* in Chapter 7.

Manipulating graphics by script

Because each graphic is an individual object, you manipulate its appearance by setting properties rather than using the drag commands (as with the paint tools, above). You can control all properties of the graphic object by script including the rectangle, line and fill properties. You can change a graphic from one type to another (e.g. a rectangle to an oval) by setting its **style** property.

The polygon style of graphic has a **points** property which allows you to set the individual points of a line.

Simple motion can be applied using the **move** command. For example, to move a graphic 100 pixels to the right asynchronously:

```
move graphic 1 relative 100,0 without waiting
```

For more information, see the **move** command in the LiveCode Dictionary.

To program a more complex animation effect, calculate the changes to the points or rectangles and set these values using timer based messaging. The following example scales a graphic named "rectangle" down by 100 pixels over the course of 1 second.

```

local sCount
on mouseUp
    put 0 into sCount
    scaleGraphic
end mouseUp

on scaleGraphic
    add 1 to sCount
    if sCount > 100 then exit scaleGraphic
    get the rect of graphic "rectangle"
    add 1 to item 1 of it
    add 1 to item 2 of it
    subtract 1 from item 3 of it
    subtract 1 from item 4 of it
    set the rect of graphic "rectangle" to it
    send "scaleGraphic" to me in 10 milliseconds
end scaleGraphic

```

See the section on *Timer based messaging* for more information on using timers.

Working with Video

LiveCode supports playback of video with the player object. On Windows *DirectShow* is supported. Media format support in the new Windows player control depends on which codecs are installed. A list of the [file formats and compression types available as standard.aspx](#)) on Windows is available in the MSDN documentation

On Mac OS X systems, the player object uses the AV Foundation framework.

On Linux Systems, the player object can play back video using *mplayer*. There are some functionality limitations: the **alwaysBuffer**, **startTime**, **endTime**, **showController** and **playSelection** properties have no effect, and **play step forward/play step back** do not work reliably.

In addition to these features, LiveCode has built-in support for the animated GIF format. Animated GIF files can be played back without 3rd party software. See above for more information. Other formats supported by plug-ins in web browsers can be played back using revBrowser (e.g. Flash). See the topic on *revBrowser* for more information.

The Player Object

Use the player object to play and interact with video and audio. To create a player object, drag one onto your stack from the Tools palette. To select a movie file to play, open the Inspector for the player object and select a file to use as the source. Doing this sets the player's **fileName** property.

To *stream* a movie from an Internet server, set the fileName property to the URL address of the stream.

To play a player, use the **start** and **stop** commands.

```

start player 1
stop player 1

```

The following table describes commonly used player properties:

Property Name	Function	Example
alwaysBuffer	Forces the player to be buffered, allowing objects to be drawn on top and the current frame to be printed	<pre>set the alwaysBuffer of player 1 to true</pre>
showController	Shows or hides the player controller	<pre>set the showController of player 1 to false</pre>
currentTime	Sets the current frame	<pre>set the currentTime of player 1 to 1000</pre>
duration & timeScale	The duration of the movie and the number of intervals per second of a movie	<pre>put (the duration of me/the timeScale of me) into tRunTime</pre>
currentTimeChanged	Message sent when the current frame changes	<pre>on currentTimeChanged pInterval put pInterval into field "Time Code" end currentTimeChanged</pre>
startTime	The start time of the selection	<pre>set the startTime of player 1 to 500</pre>
endTime	The end time of the selection	<pre>`set the endTime of player 1 to 1000</pre>
showSelection	Show the selection in the controller	<pre>set the showSelection of player 1 to true</pre>
playSelection	Play only the selection	<pre>set the playSelection of player 1 to true</pre>
playRate	The speed to play the movie. Set this to -1 to play backwards	<pre>set the playRate of player 1 to 2</pre>
looping	Causes playback to loop	<pre>set the looping of player 1 to true</pre>
playLoudness	Set the volume	<pre>set the playLoudness of player 1 to 50</pre>
tracks	List of tracks within the movie	<pre>put the tracks of player 1 into tTracksList</pre>
enabledTracks	Enable or disable tracks	<pre>Set the enabledTracks of player 1 to 3</pre>
callbacks	A list of messages to be sent when the movie reaches specified time points	<pre>set the callbacks of player 1 to "1000, nextScene"</pre>

The following properties can be used to control a QTVR movie: **pan**, **tilt**, **zoom**, **currentNode**, **nodeChanged**, **hotspots**, and **hotSpotClicked**.

For more information on any of these terms, see the *LiveCode Dictionary*.

Working with Sounds

In addition to playing back sound in a wide variety of formats using the player object, LiveCode has in-built support for playback of WAV, AIFF and AU format audio clips.

Note: We recommend you use the player object for playing audio as it supports a wide range of formats and compression types. Use the audio clip when you need basic audio playback on systems that do not have any of the 3rd party libraries supported by the player object installed.

Importing an Audio Clip

To import an audioClip choose File -> Import as Control -> Audio File. This will import the selected audio file into the current stack as an audioClip.

Playing an Audio Clip

To play an audio clip:

```
play audioClip 1
```

To stop playing

```
play stop
```

To set the volume:

```
-- sets the volume to 50%
set the playLoudness of audioclip 1 to 50
```

Working with Visual Transition Effects

LiveCode supports visual transition effects when changing card or hiding and showing objects. There are three types of effect support: built-in effects which work on all platforms, QuickTime effects which work on older Mac OS X systems and Core Image effects which work on Mac OS X.

Use the **visual effect** command to display a visual effect. To go to the next card with a dissolve transition effect:

```
visual effect dissolve slow go next card
```

To make changes to objects on the screen with a visual effect (e.g. hide, show or move them), first lock the screen, then make the changes, then unlock the screen:

```
lock screen
hide image 1
show image 3
unlock screen with visual effect "wipe right"
```

To choose a QuickTime effect using the effect chooser dialog use:

```
answer effect
-- store the visual effect as a custom property
set the cEffect of this stack to it
```

Then:

```
visual effect (the cEffect of this stack)
go next card
```

For more information on visual effects, see the **visual effect** command in the *LiveCode Dictionary*. To try out the different visual effects available in an interactive format see the [Multimedia Workshop](#)

Creating Custom Skins

In addition to its support for system native controls, LiveCode allows you the ability to create an entirely custom look, or skin, for your application. All of the built-in elements can be replaced with themed graphics allowing you to create rich multimedia. For example, LiveCode buttons support "icons" of unlimited size which can be used to replace the native appearance of a button entirely. Windows can be irregularly shaped and even contain holes and alpha mask (variable) transparency. All of LiveCode's objects support a variety of transfer modes or inks. Stacks can take over the entire screen or be displayed with a backdrop.

Custom Themed Buttons

To create a custom button, first create a regular button. Open the *Inspector* for the button and set its style to *Transparent*. Turn off the *Show Name* property. Next, switch to the *Icons & Border* pane in the Inspector. Turn off the *Hilite Border* property. You can now select any image you have imported to use as an icon for the button. To set the mouse down state, set the *hilite* icon. To set the roll over state set the *hover* icon.

A button "icon" may be an image of any size in LiveCode, allowing a completely custom look.

Tip: To use the same set of images as icons throughout your stack file, create a substack and import all your theme images into it. Set the ID property of each image to a high value (between 10,000 and 100,000) so that it will be unique. You can now reference that image anywhere in your application. If you want to change theme, you only need to replace that image with another image and give it the same ID.

Any object in LiveCode can behave like a button. For example, if you want to create a graphic that responds when a user clicks on it, create the graphic and add a **mouseUp** handler to it, in the same way you would with a button.

Irregular Windows

To create an irregularly shaped window, import or create an image that has a transparency mask. Then use the *Stack Inspector* to choose that image as the stack's *Shape*. To change the shape by script, set the **windowShape** property. Many modern window managers support alpha blended windows (variable degrees of transparency). To create a window with an alpha channel, import a PNG that contains an alpha channel and set the **windowShape** to this image.

Blend Modes (transfer modes or inks)

Blend modes determine how an object's colors combine with the colors of the pixels underneath the object to determine how the object's color is displayed. To set the blend mode of an object, use the *Blending* pane in the *Inspector* or set the object's **ink** property. All objects in LiveCode support blend modes, with the exception of stacks.

```
set the ink of image "picture" to "blendBurn"
```

For more information, see the **ink** entry in the LiveCode Dictionary.

To set the degree of transparency, set the object's **blendLevel** property. All LiveCode objects (including stacks) support **blendLevel**:

```
set the blendLevel of button 1 to 50
-- sets a button to 50% transparent
```

Full screen mode

A stack can be displayed as full screen by setting its **fullScreen** property to true:

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

Set this property back to **false** to exit full screen mode.

If you want to hide or show the menu bar on Mac OS X use the **hide menuBar** or **show menuBar** commands.

Similarly, use **hide taskbar** and **show taskbar** on Windows systems to hide and show the taskbar.

Displaying a Backdrop

To display a backdrop set the **backDrop** global property. You may set the backDrop to a solid color or to the ID of an image.

```
set the backDrop to "black"
```

To remove the **backDrop**:

```
set the backDrop to none
```

The LiveCode IDE

Introduction

This guide details the main components within LiveCode's Integrated Development Environment (IDE). The development environment contains all the features you need to quickly create a professional application. The Application Browser allows you to find your way around your application as you develop it. The Properties Inspector allows you to set appearance and basic behaviors. The Script Editor allows you to add code to each object in your application. In addition to these standard tools, the Message Box provides a mini command-line that allows you to develop aspects of your application automatically or to test out your application's code and functionality.

The Menu Bar

The File Menu

The File menu contains commands to open, close and save files; print; and incorporate files into your stack.

Menu Item	Action
New Stack	Opens a cascading menu you can use to choose a stack type to create. When you save the stack, LiveCode asks for a file name and location.
Default Size	Creates a stack at the default size (400x400)
Predefined Size	Creates a stack at the predefined size specified
Script only Stack	Creates a stack to be used for script only. You will be prompted for a stack name, and the script editor for the new stack will launch automatically
New Substack of (main stack name)	Creates a new untitled stack in the same file as the active main stack. When you save the substack, it is saved in the main stack's file. This item is disabled if the active window is not a main stack.
Open Stack...	Opens the main stack whose file you select. If you select a HyperCard file, it is automatically converted into a LiveCode main stack.
Open Recent Stack	Opens a cascading menu containing the names of the 30 main stacks you have most recently closed. Choose a name to open the stack. You can change the number of items displayed using Preferences.
Close	Closes the active window. This item is disabled if no window is open.
Close and Remove from Memory...	Closes the current stack and all stacks in the same stack file as the current stack, and removes all stacks in the file from memory. This item is disabled if the active window is not a user stack.
Import as Control	Opens a cascading menu you can use to choose a file and place its contents in a new control of the appropriate type. This item is disabled if the active window is not a stack.
Image File...	Imports the picture file you choose as a new image on the current card. You can import GIF, JPEG, PNG, BMP, XWD, XBM, XPM, or PBM, PGM, or PBM files (and PICT files on Mac OS and OS X systems). For more details see the section on the <i>image</i> object.

Menu Item	Action
Audio File...	Imports the sound file you choose as a new audio clip in the current stack. You can import WAV, AIFF, or AU files. For more details, see the section on the <i>audioClip</i> object.
Video File...	Imports the video file you choose as a new video clip in the current stack. You can import AVI, or MPEG files. For more details, see the section on the <i>videoClip</i> object.
Text File...	Imports the text file you choose as a new field on the current card.
Snapshot	Displays a crosshairs cursor for you to select an area of the screen, and imports a screen shot of that area as a new image on the current card.
All Images in Folder...	Imports all the picture files in the folder you choose, and places them in new images on the current card. Subfolders, and other types of files, are ignored. For more details see the section on the <i>image</i> object.
All Audio Files in Folder...	Imports all the sound files in the folder you choose, and places them in new audioClips in the current stack. Subfolders, and other types of files, are ignored. For more details, see the section on the <i>audioClip</i> object.
All Video Files in Folder...	Imports all the video files in the folder you choose, and places them in new videoClips in the current stack. Subfolders, and other types of files, are ignored. For more details, see the section on the <i>videoClip</i> object.
All Text Files in Folder...	Imports all the text files in the folder you choose, and places them in new fields in the current stack. Subfolders, and other types of files, are ignored.
New Referenced Control	Opens a cascading menu you can use to select a file to reference (i.e. link) to a new control of the appropriate type. This item is disabled if the active window is not a stack.
Image File...	Creates a new image on the current card and links to the picture file you select in the new image object. For more details see the section on the <i>image</i> object.
Video File...	Creates a new player on the current card and links to the video file you select in the new player object. For more details see the section on the <i>player</i> object.
All Images in Folder...	For each picture file in the folder you select, creates a new image on the current card and links it with one of the files. Subfolders, and other types of files, are ignored. For more details see the section on the <i>image</i> object.
All Videos in Folder...	For each video file in the folder you select, creates a new player on the current card and links it with one of the files. Subfolders, and other types of files, are ignored. For more details see the section on the <i>player</i> object.
Save	Saves changes to the current stack and to any other stacks that reside in the same stack file. If the file has not yet been saved, you specify the new file's name and location. This item is disabled if the active window is not a stack.
Save As...	Saves the current stack, along with any other stacks that reside in the same file, to a new file with a name and location you specify. The new file becomes the current working copy.
Move Substack to File...	Saves the front most substack as a main stack in a file of its own, with a name and location you specify. The substack is removed from its previous stack file. This item is disabled if the active window is not a substack.
Revert to Saved...	Throws away any changes to the current stack, along with any other stacks that reside in the same stack file. The entire stack file is then reloaded.
Standalone Application Settings...	Settings for packaging the current stack for distribution as a standalone application. This item is disabled if the active window is not a stack. For more details, see the section on <i>Deploying your Application</i> .
Save As Standalone Application...	Packages the current stack into a standalone application for distribution using the settings set using the Standalone Application Settings window.

Menu Item	Action
Page Setup...	Opens the Page Setup dialog box for the currently selected printer.
Print Card...	Prints the current card.
Print Field...	Prints the currently selected field using the revPrintField command. For more details, see the section on <i>Printing</i> .
Exit	Closes all open stacks and quits LiveCode.

The Edit Menu

The Edit menu contains commands to select, cut, copy, and paste text and objects.

Menu Item	Action
Undo	Reverses the most recent text change, paint action, movement or deletion of an object.
Cut, Copy, Paste	Cuts, copies or pastes the selected text or object. Text and images can be exchanged with other programs in this manner. Objects are only available within the currently running instance of LiveCode.
Clear	Deletes the selected text or objects, without placing it on the clipboard.
Duplicate	Makes a copy of the selected object or objects. If the object is a card, the copy is added after the current card. If the object is a control, the copy is placed on the current card, below and to the right of the original object. This item is disabled if no object is selected.
Select All	Selects all the text in the current field or all the controls on the current card.
Deselect All	Deselects any selected objects, or removes the insertion point from a field.
Invert Selection	Selects all the unselected objects and unselects all the selected ones. This item does not work on text selections.
Select Grouped Controls	If this item is checked, clicking a control that's part of a group selects only that control. If unchecked, clicking a control that's part of a group selects the group.
Intersected Selections	If this item is checked, dragging with the Pointer tool selects each object that intersects the dragged rectangle. If unchecked, dragging with the Pointer tool selects only objects that are entirely enclosed by the dragged rectangle.
Find and Replace...	Searches for and optionally replaces text in fields, properties, scripts, global variables, or button contents. For more details, see the section on <i>Find and Replace</i> .

The Tools Menu

The Tools menu contains commands to work with LiveCode's tool palettes and to use stack development tools.

Menu Item	Action
Browse Tool	Chooses the Browse tool for running a project. Allows you to perform user actions such as running scripts by clicking buttons, or entering text in a field.
Pointer Tool	Chooses the Pointer tool for editing a project. Allows you to select, move, and resize objects.
Tools Palette	Shows or hides the Tools palette for choosing tools for object creation.

Menu Item	Action
Paint and Draw Tools	Shows or hides the paint and draw tools in the Tools palette.
Project Browser	Opens the Project Browser window, which lists all open stacks, the cards in each stack, and the controls on each card. For more details, see the section on the <i>Project Browser</i> .
Message Box	Shows or hides the Message Box. The Message Box is a command line tool that allows you to run scripts or perform automated editing operations. For more details, see the section on <i>The Message Box</i> .
Extension Manager	Shows or hides the Extension Manager. The Extension Manager is a stack which shows what widget and library extensions are currently installed.
Extension Builder	Shows or hides the Extension Builder. The Extension Builder is a stack which aids in the creation of widget and library extensions to LiveCode. For more information on creating extensions using the Extension Builder, please see the <i>Extending LiveCode</i> guide.
Menu Builder	Creates or changes the menu bar in the currently active stack. For more details, see the section on the <i>Working with Menus</i> .

The Object Menu

The Object menu contains commands to change the properties of the selected object or objects, to create new objects, and to work with groups.

Menu Item	Action
Object Inspector	Opens the Inspector for the currently selected object, allowing you to view and set object properties. If more than one object is selected, changes made to the properties are applied to each of the selected objects. This item is disabled if no object is selected. For more details, see the section on <i>The Properties Inspector</i> .
Card Inspector	Opens the Inspector for the current card, allowing you to view and set card properties.
Stack Inspector	Opens the Inspector for the current stack, allowing you to view and set stack properties.
Object Script	Opens the Script Editor for the selected objects. If more than one object is selected, a script editor window opens for each one. For more details, see the section on <i>The Script Editor</i> .
Card Script	Opens the Script Editor for the current card.
Stack Script	Opens the Script Editor for the current stack.
Group Selected	Makes the selected objects into a group. This item changes to "Ungroup Selected" if the only selected object is a group. For more details, see the section on <i>Groups and Backgrounds</i> .
Ungroup Selected	Makes the selected group into individual objects. This removes the group permanently if you visit another card before grouping the objects again. This item changes to "Group Selected" if more than one object is selected.
Edit Group	Enters group-editing mode to make changes to the objects in the selected group. This item changes to "Stop Editing Group" while in group-editing mode. This item is disabled if no object is selected, if more than one object is selected, or if the selected object is not a group.

Menu Item	Action
Stop Editing Group	Leaves group-editing mode. This item changes to "Edit Group" if the stack is not already in group-editing mode.
Remove Group	Removes the selected group from the current card, without deleting it from the stack.
Place Group	Opens a cascading menu containing the names of groups that are in the stack but not on the current card. Choose a group to add it to the current card. This item is disabled if all the groups in the stack already appear on the current card, or if there are no groups. Note: Only the names of top-level groups are listed; groups that are a part of another group are not listed in this menu.
New Card	Creates a new card following the current card. Note: If there are any shared groups on the current card when you choose this menu item, they are automatically placed on the new card. If a group's <code>backgroundBehavior</code> is false, they are not placed automatically on new cards.
Delete Card	Deletes the current card from the front most stack.
New Control	Opens a cascading menu you can use to create a new control. For more details, see the section on <i>Control Types</i> .
New Widget	Opens a cascading menu you can use to create a new widget control. For more details, see the section on <i>Control Types</i> .
Flip	Opens a cascading menu you can use to change the orientation of the selected image or graphic. This item is disabled if any object other than an image or graphic is selected.
Horizontal	Swaps the left and right edges of the selected image or graphic, flipping it around an imaginary line drawn from top to bottom of the object.
Vertical	Swaps the top and bottom edges of the selected image or graphic, flipping it around an imaginary line drawn from left to right of the object.
Rotate By...	Opens a cascading menu you can use to rotate the selected image or graphic.
90° Right	Rotates the selected image or graphic by the number of degrees you specify.
90° Left	Rotates the selected image or graphic 90 degrees to the right (clockwise).
180°	Rotates the selected image or graphic 90 degrees to the left (counterclockwise).
Reshape Graphic	Rotates the selected image or graphic 180 degrees (halfway around).
Align Selected Controls	Allows you to interactively reshape currently selected polygon or curve graphics.
Left	Opens a cascading menu you can use to line up objects. This item is disabled if no object or only one object is selected.
Right	Moves the selected controls so their left edges are lined up with the left edge of the first control selected.
Top	Moves the selected controls so their right edges are lined up with the right edge of the first control selected.
Bottom	Moves the selected controls so their top edges are lined up with the top edge of the first control selected.
Make Widths Equal	Moves the selected controls so their bottom edges are lined up with the bottom edge of the first control selected.
	Resizes the selected controls so that the width of each one is equal to the width of the first control selected.

Menu Item	Action
Make Heights Equal	Resizes the selected controls so that the height of each one is equal to the height of the first control selected.
Send to Back	Moves the selected objects behind all other objects on the card. This item is disabled if no object is selected.
Move Backward	Moves the selected objects back one layer.
Move Forward	Moves the selected objects forward one layer.
Bring to Front	Moves the selected objects in front of all other objects on the card.

The Text Menu

The Text menu contains commands to change the appearance of text.

Menu Item	Action
Text Style Options	Applies or removes the selected style from the selected text or the selected object. These items are disabled if nothing is selected.
Link	Makes the selected text, or all text in the selected object into a link. Linked text has special properties in LiveCode, for example linked text will receive a linkClicked message when clicked on. For more details, see the section on the <i>Field</i> object.
Subscript	Moves the selected text below the baseline and makes it smaller.
Superscript	Moves the selected text above the baseline and makes it smaller.
Font	Opens a cascading menu you can use to change the font face used for the selected text or objects.
Use Owner's Font	Causes the font face of the object's owner (if an object is selected) or the object the text is in (if text is selected) to be used, removing any font face specific to the selected text or objects. For more information on font inheritance, see the section on the <i>Message Hierarchy</i> .
Size	Opens a cascading menu you can use to change the font size used for the selected text or objects.
Use Owner's Size	Causes the font size of the object's owner (if an object is selected) or the object the text is in (if text is selected) to be used, removing any font size specific to the selected text or objects. For more information on font inheritance, see the section on the <i>Message Hierarchy</i> .
Color	Opens a cascading menu you can use to change the font color used for the selected text or objects. (This item is disabled if nothing is selected.)
Use Owner's Color	Causes the font color of the object's owner (if an object is selected) or the object the text is in (if text is selected) to be used, removing any font color specific to the selected text or objects. For more information on font inheritance, see the section on the <i>Message Hierarchy</i> .
Pen Color	Changes the selected text, or text used in any selected objects, to use the current 'pen color' setting (used to draw graphics from the main tool palette).
Align	Opens a cascading menu you can use to change the text alignment (justification) used for the text in the selected objects.

The Development Menu

The Development menu contains commands for debugging and for using custom tool stacks.

Menu Item	Action
Plugins	Opens a cascading menu you can use to open custom tool stacks stored in the Plugins folder.
Plugin Settings	Customizes which messages are handled by the custom tool stacks stored in the Plugins folder.
Test	Simulates deployment to the selected test target.
Test Target	Opens a cascading menu you can use to select what device or simulator to use when simulating deployment.
Script Debug Mode	If this item is checked, the debugger is enabled: the debugger window appears when a breakpoint is encountered during script execution, and you can enter the debugger when an execution error occurs. If unchecked, the debugger is disabled. For more details, see the section on <i>Debugging</i> .
Clear All Breakpoints	Removes all breakpoints that you've used the Script Editor to mark, in all open stacks. Note: This menu item does not affect breakpoints set with the <i>breakpoint</i> command.
Message Watcher	Opens the Message Watcher window, used to view and track messages, function calls, getProp calls, and setProp triggers as they are sent to objects.
Suppress Errors	Prevents display of the error window when LiveCode encounters a script error.
Suppress Messages	Prevents system messages (such as openCard and closeCard) from being sent during normal navigation. This option will also cancel any currently <i>pending</i> messages.
Suspend Development Tools	Hides LiveCode's menus, palettes, and other parts of the development environment, so that you can preview how your application will look and behave as a standalone, outside the development environment.

Caution: Suppressing errors will cause your scripts to fail silently instead of giving an error message in the event LiveCode encounters an error.

Tip: To also suspend LiveCode libraries, hold down the Shift key while choosing the **Suspend Development Tools** menu item.

The View Menu

The View menu contains commands to move around the current stack and to show or hide development tools.

Menu Item	Action
Go First	Goes to the first card in the current stack.
Go Prev	Goes back to the previous card in the current stack.
Go Next	Goes forward to the next card in the current stack.
Go Last	Goes to the last card of the current stack.
Go Recent	Goes back to the card you were on before navigating to the current card.
Toolbar Text	Shows or hides the text labels in the Toolbar at the top of the screen. To hide the Toolbar completely, uncheck both this item and "Toolbar Icons".
Toolbar Icons	Shows or hides the icons in the Toolbar at the top of the screen.
Palettes	Shows or hides all open LiveCode palettes.
Rulers	Shows or hides a ruler at the left and bottom edges of each open stack.

Menu Item	Action
Grid	If this item is checked, dragging and resizing objects is constrained by a pixel grid. If unchecked, you can drag and resize objects to any location. You can alter the grid spacing in Preferences.
Backdrop	Shows or hides a solid or patterned backdrop behind LiveCode's windows.
Show IDE Stacks in Lists	If this item is checked, elements of the LiveCode development environment appear in lists: for example, development environment stacks appear in the Application Browser, and LiveCode custom properties appear in the Custom Properties pane of the property inspector. If unchecked, elements of the LiveCode development environment do not appear in such lists.
Show Invisible Objects	If this item is checked, objects whose visible property is set to false are shown. If unchecked, objects whose visible property is set to false remain hidden.

The Window menu

The Window menu contains the names of open stack windows.

Menu Item	Action
Send Window To Back	Shuffles the frontmost stack window to the back and brings the second window to the front. (This item is disabled if only one stack is open or the active window is not a stack.)

The Help Menu

The Help menu contains commands which allow you to find out more about using LiveCode, license your copy of LiveCode, get technical support via email, and check the LiveCode site for software updates. For more information, see the section on *Using the Documentation*.

Menu Item	Action
Dictionary (API)	Opens the main LiveCode documentation window, where you can get information about all aspects of developing in LiveCode.
Start Center	Opens the Start Center.
Sample Stacks	Launches revOnline in an external browser
Sample Scripts	Opens the Resource Center, which contains many sample scripts, including tutorials on text processing and using databases.
Beginner's Guide	Opens the beginner's guide in an external browser
All Guides	Opens the main LiveCode documentation window, which contains all of the user guides
Tutorials	Launches the online LiveCode lessons portal in an external browser
User Guide	Opens the old user guide in an external PDF viewer
Forums	Navigates to the LiveCode discussion forums in an external browser
Technical Questions	Navigates to questions tagged LiveCode on StackOverflow in an external browser
Discussion List	Navigates to the LiveCode discussion list in an external browser
Newsletters	Navigates to LiveCode news letters and blogs in an external browser

Menu Item	Action
Release Notes	Opens the release notes in an external PDF viewer
License Agreement	Opens the license agreement in an external text editor
User Guide	Opens the old user guide as a PDF document
Support...	Loads the support section of the LiveCode web site in your web browser.
Buy a license...	Loads the LiveCode store in your web browser.
Check for Updates...	Checks online to see if an update to LiveCode is available.
Re-license LiveCode	Deletes your LiveCode licensing information and restarts LiveCode allowing you to enter a new license key.

The Property Inspector

The Property Inspector allows you to view and edit the properties for any selected object. Properties control how an object looks and some aspects of an object's behavior. The Inspector can be accessed by double clicking on a selected object, from the toolbar, from the *Object* menu and from context sensitive menus.

Figure 9 - Properties Inspector

Lock icon	The lock icon locks the Inspector to the object currently being inspected. Once locked, the Inspector will not update when you change selection or switch to <i>Run</i> mode by choosing the <i>Browse tool</i> . This allows you to use a locked Inspector to change an object's properties while interacting with your project. Selecting another object and choosing <i>Inspect</i> will create another Inspector to inspect the new object. You can use this facility to compare the properties of two objects.
Pane Selector	This menu allows you to access each of the different panes within the inspector for a given object.
Action menu	Use the action menu to select another object, edit the script of the current object or send a message to current object.
Settings menu	Use the settings menu to configure the settings of the property inspector.
Property text	Edit the contents of a property by typing into the text area within the Inspector. Press return to set the property and leave the current option selected. Press tab to set the property and move on to the next editable text property.
Property description	This text describes the object property. By default this text is an English-like description of the property. However you can change this to contain the name of the LiveCode property in the Preferences or the settings menu. You may prefer to view the LiveCode property names if you are writing scripts that set object properties.

Important: For documentation on what each individual object property does, hover the mouse over the object until you see the script property equivalent. Then look that term up in the LiveCode Dictionary.

Property Inspector settings

Tab display style	Controls whether to use icons or text for the pane selections.
Header/Footer size	Controls how large the pane selector icons are.
Property labels	Controls whether to use the LiveCode script tokens for property labels or not.

The Script Editor

The Script Editor within LiveCode has been designed specifically for LiveCode coding. It includes features to help make code more understandable. These include code indentation and color coded syntax highlighting, as well as other integrated tools such as a Debugger and syntax Dictionary. You can access the Script Editor for an object by selecting the object then choosing Script from the Tool bar. The Script Editor is also available from the Object menu, and from a number of other context sensitive menus detailed elsewhere in this guide. Each of the Script Editor's components are outlined below.

Main Script Area

Display and edit scripts in this area. For more details on how to write a script, see the section on Coding in LiveCode. The Script Editor will colorize and format scripts automatically. When typing press tab to manually format the script. Press Enter (on the numeric keypad) to compile the script. Press Enter again to close the Script Editor.

Note: It is not necessary to close the Script Editor to run a script, all that is required is that the script is applied using the Apply button.

Breakpoints Area

Click next to a line of script to set or remove a breakpoint. A breakpoint specifies a point at which script execution should pause and the debugger be loaded. For more details, see the section on Debugging.

Apply Button

Compiles the script instantly. Any syntax errors will appear in the Error Display and the script will not compile. A compiled script is not saved until the stack that contains it is saved.

Press *Enter* (on the numeric keypad) to activate the Compile button. Press *Enter* again to close the Script Editor.

Note that *none* of the message handlers within the script will be available if there is a Script Error during compilation, as none of the script will have compiled.

Errors Pane

Compile and execution errors will appear here. Double click the icon next to the error details to highlight the line the error occurred on in the script (if applicable).

Handler List

The handler list displays all the functions, commands and handlers which are part of the current script. When a handler name is clicked, the script will jump to that handler.

The currently displayed list of handlers can be filtered by typing in the filter box below it.

Documentation Pane

The Script Editor also has a built-in syntax dictionary. When this tab is active a summary of the dictionary entry for the keyword which you are currently typing will be displayed. The full dictionary entry can be viewed by clicking Launch Documentation. Alternatively you can choose to view the entire entry in place by toggling the Full Document check box at the bottom of the documentation pane.

Search Results Pane

When you perform a 'find all' operation the results will be displayed under this tab.

Script Tabs

When multiple scripts are open they will appear as tabs in the Script Editor. Right clicking on the tabs brings up a menu which allows a tab to be moved to a new window, closed, or kept while closing all other tabs. Clicking a tab will bring up the script it represents.

The Debugger

The Debugger helps you track bugs and understand code, by allowing you to pause the execution of your programs or step through them line by line. You cause execution to pause by setting breakpoints. When you run your script, execution will pause at a breakpoint and you will be able to observe or edit variable values, continue execution or set another breakpoint later in the code. Each debugger feature is outlined below:

Continue

When in debug mode, the Continue button will start running the code from the current position. The program will run until the end is reached or another breakpoint is found.

When you are not in debug mode, the continue button can be used to execute a handler. When you press it, a dialog will appear asking which handler you would like to call and, if applicable, which parameters you'd like to pass. On clicking OK the Script Editor will call the handler you specified. It will also remember the handler for next time. To change the handler called later, choose Entry Point... from the Debug menu.

Stop Debugging

Will halt execution at the current point and stop debugging. You will then be able to edit the script.

Show Next Statement

The Show Next statement option will return the debugger to the currently executing statement. This is useful if you have switched tabs or scrolled during debugging.

Step Into Next Statement

Use Step Into to execute the next statement. If the next statement is a command or function 'Step Into' will jump to that function or command allowing you to execute it line by line.

Step Over Next Statement

Use Step Over to execute a command or function call without stepping through it line by line. The code within the handler will run at full speed and execution will pause on the line after the handler call.

Step Out

The Step Out function allows you to exit a command or function that you have previously 'stepped into'. When selected, the rest of the current handler will run and execution will pause on the line after the handler call. This is useful to avoid stepping through a long command or function line by line when you have identified that the error you are looking for is not within that command or function.

Debug Context

The Debug Context shows you the path of execution up to the current statement you are paused on in the debugger. This allows you to identify where you are in your program and which handlers called the current handler you are in. Changing the debug context allows you to view variables from different parts of the program. Use this feature to find an erroneous command call or a call where wrong parameters were used.

Variable Watcher

The Variable Watcher tab allows you to examine the value of variables within your program while it is executing. As you step through your code these values will be updated. The variable name is shown in the left column and its value adjacent on the right. If a variable's value is too large to be displayed in the pane it will have a magnifying glass next to it. Clicking this will bring up a watch window which allows the variable's value to be changed. Variables can also be edited by double clicking on their value in the variable watcher.

Tip: You can see the value of a variable by moving the mouse over the variables' name in the script field while debugging.

Breakpoint Management

Use breakpoints to specify where to pause execution in the debugger. You set breakpoints by clicking in the gutter, or by right clicking in the Breakpoint Manager at the bottom of the Script Editor and selecting "New Breakpoint". Each breakpoint is associated with a line of code and execution is paused when that line is reached. Alternatively, a condition can be assigned to the breakpoint. In this case, execution will only pause if the condition is true when the line is reached.

Tip: To see a list of all the breakpoints within a stack, including the script they are in, which line they are on, and whether there is a condition attached, click the Breakpoints Manager at the bottom of the Script Editor. The check box to the left of each breakpoint can be used to enable or disable the breakpoint. Double click a breakpoint to go to that line in the associated object's script. To edit a breakpoint, including adding a condition or changing the line number, click the pencil icon. This can also be done by right-clicking on the breakpoint, either in the Breakpoints Manager, or the Script Editor's gutter.

The Breakpoint Manager also allows you to set watches. These are associated with variables rather than lines of code and will pause execution when the value of the variable you are watching changes. Alternatively, if the watch has a condition attached, execution will only pause if, when the variable's value is changed, the condition is true. To add a watch, right click in the Breakpoint Manager and choose “New Breakpoint”.

The Script Editor Menubar

The Script Editor menubar contains commands to help you edit scripts.

The File Menu

The File menu contains commands to print a script or close the Script Editor, with or without saving changes to the script.

Compile	Compiles the current script.
Save	Saves the current stack.
Close	Closes the current script.
Print	Prints the current script.
Quit	Exits LiveCode. Will prompt you to save your work if applicable.

The Edit Menu

In addition to the standard commands to select, cut, copy and paste text, the Edit menu contains the following, script-specific commands:

Revert	Takes any changes which haven't been compiled, and removes them from the current script.
Comment	Places a comment character at the beginning of the selected text. If more than one line is selected a comment character is placed at the beginning of each line.
Uncomment	Removes comment characters from the selected text.
Quick Find	Opens the “find” field within the Script Editor.
Find and Replace...	Brings up the Script Editor's find and replace dialog.
Go	Brings up a dialog box which lets you enter a line number to jump to in the script.

Variable Checking

Variable checking causes LiveCode to perform a stricter check of your scripts. Using any variable without declaring it first, or using a literal string without enclosing it in quotes causes a compile error. This behavior can be useful in tracking down certain subtle problems such as misspelling a variable name.

The Debug Menu

The Debug menu contains commands to aid in debugging scripts:

Show Next

Highlights the next line of code to be executed. This is useful if you have changed tab or scrolled your script during debugging.

Step Into

Executes the next line of the handler being debugged. If the next line is a call to another handler, this command steps into that handler. This menu item is disabled if the debugger is not running.

Tip: Press F11 to step into the next line.

Step Over

Executes the next line of the handler being debugged. If the next line is a call to another handler, this command steps over that call, skipping it and staying within the current handler.

Tip: Press F10 to step over the next line.

Run

Resumes normally executing the handler being debugged, starting from the current line.

Tip: Press F5 to activate the Run button.

Abort

Stops running the handler being debugged.

Tip: Press control-period (or command-period) to activate the Abort button.

Toggle Breakpoint

Adds/removes a breakpoint from/to the currently selected line.

Variables

Changes the tab view at the bottom of the Script Editor to the Variable Watcher tab.

Breakpoints

Changes the tab view at the bottom of the Script Editor to the Breakpoint Manager.

Entry Point

Lets you define which handler execution should begin at when debugging a script.

Script Debug Mode

If this item is checked, the debugger is enabled: the execution of the program will pause when a breakpoint is encountered during script execution, and you can enter the debugger when an execution error occurs. If unchecked the debugger is disabled. For more details, see the section on Debugging.

The Handler Menu

The Handler menu contains a list of all handlers in the current script. Choose a handler to go to that handler in the script.

The Window Menu

The Window menu contains the names of open script editor windows.

Autocomplete

Using Autocomplete

While typing in the script editor a filtered list of available completions will appear below the selection. Use the following keys to navigate the completion list:

- right arrow - apply the completion or enter group of completions
- left arrow - leave group of completions
- up arrow - move up the list
- down arrow - move down the list

Moving the mouse will hide the completion list to avoid it remaining in place over an area you would like to select. Completions include placeholder fields which you can navigate with the tab key. Some completions may have multiple placeholders with the same name which will be edited together. Clicking within a placeholder will select it.

Autocomplete Pro

Autocomplete Pro provides the following additional features:

- Completions generated dynamically by introspecting the object being edited and its message path
- An Autocomplete Snippet Manager dialog is accessible from the script editor menubar to manage a custom set of completions.

Autocomplete Snippet Manager

An autocomplete snippet provides the information required to match and complete some code in the script editor. The Autocomplete Snippet Manager allows the user to manage a custom set of completion snippets.

Create a new completion snippet by clicking the + icon and delete the currently selected snippet by clicking the - icon.

The dialog allows entry of the following:

Display Name - choose a completion snippet to edit or change the display name of the current snippet.

Priority - A value between 1 and 100 indicating the order the completions should be matched in. Higher priorities will be listed before lower priorities.

Match Scope - Choose whether the completion should try and match when the selection is outside a handler, inside a handler or anywhere. A comment completion might be anywhere, a statement completion (unless local, global or constant) would be only inside a handler and a handler completion would be outside a handler.

Match Words - Choose whether to match only all the words of a line, just the word being written or either. A statement would be all the words on a line, an expression would be just the current word and a comment would be either.

Alternate Match String - When matching completion snippets the text of the completion is matched first but in some cases that is not suitable or another string would be better to match. Enter a string to try matching if the completion fails to match.

Summary - This field is optional and provides some detail below the completion in the script editor.

Completion - This is the string to be used when replacing the selection after choosing a completion. The completion may be any script and any length. Add placeholders to the completion in the following form:

```
${<name>:<type>}
```

Where:

- name - is the text seen in the script editor
- type - is used to filter the completions presented for the placeholder one of:
 - statement - any LiveCode command
 - expression - any LiveCode expression
 - identifier - a variable or handler name

Additionally, it is possible to include multiple placeholders with the same name. When a multiple placeholder is selected they are edited at the same time. In this example completion as the user enters the condition a comment is appended to the `end if` with the condition text. Once the condition is complete the user may tab to the `-- code` line and enter a statement. The tab order is always top to bottom and left to right:

```
if ${condition:expression} then
  ${-- code:statement}
end if # ${condition}
```

The Message Box

The Message Box is a command line tool that allows you to run scripts or perform automated editing operations. It allows you to try out short scripts, test parts of your program, provides a convenient output window for debugging and can be used for editing and setting properties.

Tip: The Message Box is one of the more useful components in LiveCode when you are starting out and wanting to try out simple scripts. You will find you can try out many of the script examples in this User's Guide or in the getting started materials on our website simply by copy-pasting them into the Message Box.

The message box has the following modes:

Single Line - execute single line and short scripts

Multiple Lines - execute multiple line scripts

Global Properties - view and edit global properties

Global Variables - view and edit global variables

Pending Messages - view, cancel and edit pending messages

Front Scripts - view and edit frontScripts

Back Scripts - view and edit backScripts

Figure 10 – The Message Box in Single Line Mode

Type valid LiveCode code into this area. In *Single Line* mode, separate multiple lines with ; (semi-colon) and press *Return* to execute. In *Multiple Line* mode, press return to separate lines, and press *Enter* to execute.

For example, in Single Line mode, to move all controls on the current card 10 pixels to the left you would run: repeat with i = 1 to the number of controls;

```
move control i relative -10,0;end repeat
```

In Multiple Line Mode: repeat with i = 1 to the number of controls

```
move control i relative -10,0
```

```
end repeat
```

Command area

Type **control-m** to focus the command area and start typing, whenever the Message Box does not have the focus. Press **control-u** when typing to clear the command area.

In *Single Line* mode press the **Up Arrow** key to cycle backwards through the history of previously entered and executed scripts. Press the **Down Arrow** key to cycle forwards. In *Multiple Line* mode, press **alt-up arrow** or **alt-down arrow** to cycle forwards or backwards, respectively. Press **control-u** to clear the current command area.

Typing the name of a global variable or property on its own will result in the line being auto-completed to include **put** in front of the global variable or property, thus the contents of that global will be placed into the results area. For example typing:

```
time
```

Will be auto completed to:

```
put the time
```

The current time will be placed in the results area.

Displays:

The *result* function after running code in the *command area*.

Any script that uses the **put** command without a destination container.

Any script compile error created as a result of attempting to run code placed in the command area.

Results area

Any execution error created as a result of attempting to run code placed in the command area.

The **msg** special global will be updated whenever anything is placed in the results area, either by script or by directly editing the content of the results area. You can set or retrieve the content of this variable in any script. For example, try running the following in the command area:

```
put the clipboardData; replace return with return & return in msg
```

The result placed into the results area will contain the current contents of the clipboard, with every return character replaced with a second return character.

Select a stack to work on. Prior to executing any script the *defaultStack* will be set to the stack displayed in this menu. By default the Message Box will choose the front most editable stack. The menu is updated whenever you change the front most stack or click on the Message Box. You can use this menu to choose an alternative open stack. For example, running the following script in the command area of the single line message box:

Stack selector

```
put the number of controls
```

Would place the number of controls in the current card of the *stack displayed in the menu*, into the results area.

Lock icon

This option prevents the Stack selector from updating automatically when you change stacks. Use this option if you want to run a number of commands on a specific stack regardless of which stack is currently in front.

Settings menu

Use the settings menu to configure the settings of the message box.

Target action

Click the target icon to navigate to the last line of script that executed a **put** command with the message box as its target.

Message Box settings

Tab display style

Controls whether to use icons or text for the mode selections.

Header/Footer size

Controls how large the mode selector icons and footer action icons are.

Auto update pending messages

Toggles whether the pending messages as displayed in message box in pending messages mode are updated automatically or manually.

The Message Box will attempt to auto complete the typing of a name of an object property, resulting in putting the contents of that property into the results area. You can choose whether the auto-complete should attempt to use the currently selected object, or the object directly underneath the mouse. For example, with an object selected, entering:

width

**Select
intelligence
object**

Will result in:

put the width of the selObj

The width of the currently selected object will be placed in the results area. The `selObj` will be substituted with the `mouseControl` if you choose this option. In the example above, that would result in the width of the object underneath the mouse being placed in the results area instead. For more details on the `selObj` function or the `mouseControl` functions, see the LiveCode Dictionary.

Single and Multiple Line Modes

Use the Single and Multi-Line modes to execute short scripts.

Global Properties

The Global Properties mode allows you to view and edit all global properties. Scroll to select a property from the list on the left and select it to view or edit it. Typing a portion of the property name into the Filter field at the top will filter the list of properties. Changes to global properties take effect immediately. If a property is 'read only' this will be indicated when you select it and editing it will have no effect. For more details on Global Properties, see the section on *Global Properties*.

Global Variables

The Global Variables mode allows you to view and edit all global variables. Environment variables are shown first, followed by other variables. Scroll to find a variable from the list on the left and select it to view or edit it. Typing a portion of the variable name into the Filter field at the top will filter the list of variables. Changes to global properties take effect immediately. The Show LiveCode UI Variables is an advanced option, discussed in the section on Editing the LiveCode User Interface. For more details on Global Variables, see the section on *Global Variables*.

Pending Messages

The Pending Messages mode allows you to view all the current timer based messages that are pending. This is the same as the global property the **pendingMessages**. You can select a pending message to edit its script or cancel it.

Tip: You can cancel all current pending messages using the **Suppress Messages** button on the toolbar.

Pressing update refreshes the list with the current list of pending messages. The Auto Update mode refreshes the list continuously. Note that the Auto Update takes place every 200 milliseconds. If your messages fire more quickly they may not be displayed. For more details on Pending Messages, see the section on *Timer Based Messaging*. The Show LiveCode UI Messages is an advanced option, discussed in the section on *Editing the LiveCode User Interface*.

Front Scripts & Back Scripts

These modes list all the scripts currently available as libraries at the front or back of the message path. This is the same as the global properties the **frontScripts** and the **backScripts**. Select a front or back script to remove it or edit its script. For more details on library scripts, see the section on *Extending the Message Path*. The Show LiveCode UI Scripts is an advanced option that displays all of the libraries used by the LiveCode IDE. This is discussed in the section on *Editing the LiveCode User Interface*.

Stacks In Use

This mode is the same as the Front and Back scripts mode, except that it displays the **stacksInUse** global property. You can also add a stack directly using the Add button.

The Toolbar

The main toolbar provides easy access to commonly used functions.



Figure 11 – The Main Toolbar

For details on what each toolbar icon does, see the section on the Menu Bar (above).

To hide and show the text or icons on the tool bar, use **View -> Toolbar Text** and **View -> Toolbar Icons**. To hide the Toolbar completely, uncheck both options.

Find and Replace

The Find and Replace dialog allows you to search your entire application, a portion of the application, multiple files in a directory, or stacks specified in your application's **stackFiles** property. You can search for object names, scripts, field & button text, custom properties, and other properties. After performing a search you can replace the search term with a replacement term, either in all of the results or on a selection of the results.



Figure 12 – Find and Replace

Find field Enter a search term.

A menu with the following options:

The current selection – Searches the currently selected objects

This card – Searches the current card of the front most editable stack

This stack – Searches the front-most editable stack

This stack File – Searches the mainStack and all of the subStacks within the stack file associated with the current front most editable stack

This stack file and its stack files – Includes the stacks referenced in this stacks stackFiles property. For more details on the stackFiles property, see the section on *Structuring Your Application*.

Search In menu

> **Important:** StackFiles are normally set on the mainStack of your application. Ensure that you bring the mainStack to the front before searching its stackFiles. If you search using this option when a subStack is in front, your search will not include any stackFiles.

All open stacks – Searches all open stack files that are currently on screen

All open stack files – Searches all open stacks, including all mainStacks and subStacks in each file

Specific open stack file – Allows you to select a specific open stack file to search

All stack files in a folder – Searches all the stacks in a folder you specify

Global variable contents – Searches the contents of currently declared global variables

Find button Case Sensitive

Performs the search

Specifies that the search should be case sensitive (e.g. "a" and "A" are treated as different characters)

Regular Expression

Specifies that the Find Field contains a regular expression instead of plain text. A regular expression allows you to describe a pattern of text to match. For more details on using regular expressions, see the section on *Using Regular Expressions* below.

Obey dontSearch properties Search marked or unmarked cards only

Specifies that fields, cards or groups with their dontSearch property set to true should be excluded from the search. For more details on the *dontSearch* property, see the LiveCode Dictionary.

Search in... Name Script

Allows you to specify which object properties to search:

Searches an object's *short name* property.

Searches an object's *script* property.

Custom

Searches and objects custom properties and custom property sets. For more details on Custom properties, see the section on *Custom Properties*.

Field Text

Searches within field text.

Button Text	Searches button text. This includes text in menus. For more information on buttons and menus, see the section on <i>Control Types</i> .
All Other	Searches all other commonly used object properties. For more details on exactly what is included, see the <i>properties</i> entry in the LiveCode Dictionary.
Results column header	Allows you to sort search results by clicking. Drag between the column headers to change column widths.
Search Results Area	Displays a list of objects that match the search parameters.
Remove from List	Removes the currently selected search result in the Search Results area from the list. Once removed, the object will no longer be included in any replace operation.
Replace With	Specifies a term to use to replace the search term with.
Replace in All	Replaces the Find Field with the Replace With field in all objects currently listed in the Search Results Area.

Keyboard Shortcuts

The Development Environment	Windows / Linux	Mac OS X
Choose run (browse) tool	Control-9	Command-9
Choose edit (pointer) tool	Control-0	Command-0
Toogle between run and edit tools	Control-alt-tab	Command-option-tab
Hide or show palettes	Control-tab	Command-control-tab
Display context menus when running	Control-shift-right click	Command-control-shift-click
Save all open stacks	Control-alt-s	Command-option-s
Apply default button in Save dialog	Return	Return
Apply non-default button in save dialog	Control-1 st letter of button	Command-1 st letter of button

Navigation	Windows / Linux	Mac OS X
Go to the first card	Control-1	Command-1
Go to the previous card	Control-2	Command-2
Go to the next card	Control-3	Command-3
Go to the last card	Control-4	Command-4
Go to the recent card	Control-5	Command-5
Go to the top or bottom of a field	Control-up/down arrow	Command-up/down arrow

Objects	Windows / Linux	Mac OS X
Select all	Control-a	Command-a
Duplicate	Control-d	Command-d
Create card	Control-n	Command-n
Nudge control	Arrow keys	Arrow keys
Nudge control by 10 pixels	Shift-arrow-keys	Shift-arrow-keys
Open object inspector for selection	Return	Return
Open stack inspector	Control-K	Command-K
Remove styles from selected text	Control-;	Command-;
Equalize heights of selected controls	Control-shift-=	Command-shift-=
Equalize width of selected controls	Control-=	Command-=
Magnify image with paint tool	Control-right-click	Command-click
Apply transparency with paint tool	Control-click	Control-click

Objects		Windows / Linux	Mac OS X
Constrain paint tool selection to a square		Shift	Shift
Constrain object aspect ratio		Shift	Shift
The Code Editor		Windows / Linux	Mac OS X
Edit script of selected object		Control-e	Command-e
Edit card script		Control-shift-c	Command-shift-c
Edit stack script		Control-shift-s	Command-shift-s
Edit script of object under mouse		Control-alt-click	Command-option-click
Apply changes		Enter	Enter
Apply changes and close		Enter twice	Enter twice
Apply changes and save stack		Control-s	Command-s
Comment out selected lines		Control-hyphen	Command-hyphen
Remove comments		Control-shift-hyphen	Command-shift-hyphen
Switch to find mode		Control-f	Command-f
Find next		Control-g	Command-g
Find selected text		Control-Option-f	Command-Option-f
Format handler or move to next placeholder		Tab	Tab
Next tab in Editor		Control-tab	Control-tab
Previous tab in Editor		Control-shift-tab	Control-shift-tab
Show completions palette		F1	F1
The Message Box		Windows / Linux	Mac OS X
Open/close message box		Control-m	Command-m
Switch to first tab in message box		Control-m	Command-m
Clear message field		Control-u	Command-u
Scroll through recent messages (single line)		Up/down arrow	Up/down arrow
Scroll through recent messages (multiple line)		Alt-up/down arrow	Option-up/down arrow
Execute message (single line)		Return	Return
Execute message (multiple line)		Enter	Enter
		Control-Return	Control-Return
The Debugger	Windows / Linux	Mac OS X	
Step Into	F11	F11	
Step Over	F10	F10	
Step Out	Shift-F11	Shift-F11	
Run	F5	F5	
Stop	Shift-F5	Shift-F5	
Abort	Control-.	Command-.	
The Project Browser	Windows / Linux	Mac OS X	
Scroll through objects	Up/down arrow	Up/down arrow	
Expand/contract child objects	Right/left arrow	Right/left arrow	
Open object	Return	Return	
Edit script of object	Shift-Return	Shift+Return	
Edit behavior script of object	Shift-Alt-Return	Shift-Alt-Return	
Inspect object	Control-Return	Command-Return	

Deploying Your Application

Introduction

With LiveCode, it is easy to deploy your application to anyone.

Using the standalone building capability in LiveCode you can create a native application for each operating system you want to support. Users who do not have LiveCode can run these applications like any other application they download and install. Standalone applications can have their own identity as true applications, include icons, document associations and more.

Building a Standalone Application

When you have finished your LiveCode application and what to distribute it you can build it into a standalone application. These applications do not require users to have LiveCode. All of LiveCode's feature set is available for use in a standalone application, with the exception that you cannot set scripts on objects.

The builder itself will let you build standalone applications for any platform it supports, from any platform it supports with the exception that iOS standalones must be built on Mac OS X. For example you can build a Windows standalone on a Mac OS X machine. However, you may wish to check that your application looks and behaves correctly on each platform you intend to support. Please note it is inherently harder to debug an application that has been built as a standalone, so you should test your application as thoroughly as possible before building it.

Related lessons:

- [Building Standalone Applications](#)
- [How do I Develop Cross-Platform in LiveCode?](#)

Step by step

Deploying an app to standalone is straightforward:

- 1) Open your stack in the LiveCode IDE
- 2) Select **File → Standalone Application Settings...** from the menu bar
- 3) Select the settings for your app
- 4) Make sure that checkboxes for each platform you wish to build for are checked
- 5) Close the standalone settings window
- 6) Select **File → Save as Standalone Application...** from the menu bar

Your application will be packaged up and placed in the selected output folder.

Standalone Applications Settings

The Standalone Applications Setting dialog allows you to create settings for your standalone application. This dialog can be found in the File menu. The settings you enter are applied to the current front most editable stack and are saved with the stack. This means you only need to enter the settings once for each application you create. The same settings will apply if you do another build in the future.

Related lessons:

- [The Standalone Application Settings](#)

General Settings

Figure 1 – Standalone Settings – General Tab

Mode Selector	Choose between the different standalone application settings screens.
Standalone Name	Set the name of your standalone application. This should be the name you want your finished application to have. Don't include a file extension (.exe on Windows or .app on Mac OS X) as the standalone builder can create standalones for multiple platforms and will add the appropriate extension automatically.
Inclusions Selector	Choose the components you want to include in a standalone. You may either choose to search for required inclusions automatically, or manually select the components you want to include.

Search for Inclusions

This is the default option. When selected, LiveCode will search your application stack file (main stack and sub stacks) to attempt to determine what components your application uses. It will then include those items.

Select Inclusions for the Standalone Applications

Select this option if you want to specify the components to include manually. You may wish to use this option if your application dynamically loads components that cannot be searched at this point automatically, or if you know exactly what components your application uses and wish to speed up the standalone building process by skipping the automatic search step.

It is important that you choose to include all the components that your application uses or it may fail. If you do not include your own custom error reporting or LiveCode's standalone error reporting dialog (discussed below) such failure may be silent – i.e. an operation in your standalone will simply cease working without displaying anything to the user. |

Profiles Settings	Choose between the Property Profile settings options. You only need to alter settings in this area if you have used Property Profiles (see the section on <i>Property Profiles</i> in Chapter 4, <i>Builder a User Interface</i> above)
Remove all profiles	Removes all profiles and builds the standalone using the currently active profile on each object. Select this option if you don't need to change profile in the standalone and want to save disk space by removing extraneous profile information.
Set all objects to profile	Set all objects to a specific profile then remove the profile data from objects.
Include profiles and the profile library	Include the profile library and allow switching between profiles in the standalone application. You can choose whether to include specific profiles or all profiles.

Including Additional Stacks

Figure 2 – Standalone Settings – Stacks Tab

Stack Files	Use this section to add additional stack files to your application. Any stacks you add to this section will be added to the <code>stackFiles</code> property of the main stack of your standalone. This means that any scripts within your standalone application will be able to locate and reference these stacks by name.
Advanced Options	Use this section to control exactly how multiple stack files are managed in your standalone.
Move substacks into individual files	If you select this option, each of the sub stacks in the stack files you select will be moved into their own individual file, located in the data folder or within the application bundle of your standalone.
Rename stackfiles generically	Renames each sub stack file using a number on disk (instead of using the name of the sub stack). Select this option if you do not want the names you have selected for stacks to be visible to the end user in the filing system.
Create folder for stackfiles	Creates a folder and places the stack files into that folder, instead of storing them at the same level as the standalone executable. All references in the <code>stackFiles</code> property will refer to this folder using a relative path so the stacks can still be located by the standalone application.
Individual stack options	Select a stack file on the left then an individual stack from within the file to set options on that stack.
Set destroyStack to true	Set this option if you want the selected stack file to be removed from memory when it is closed. This option is useful if you are loading large stacks into memory and want them to be removed when they are closed.
Encrypt with password	Secures the scripts within the selected stack file with a password. This provides a basic level of encryption that prevents someone from casually reading the scripts in the stack by opening the file in a binary file viewer.

Note: A stack file directly attached to a standalone application cannot have changes saved to it. This stack is bound directly to the executable file that runs. The OS locks an executable file while it is running. If you want to save changes in your standalone application, split your stack up until multiple files. A common technique is to create a "splash screen" stack that contains a welcome screen and then loads the stacks that make up the rest of your application. These stacks are referenced as `stackFiles` on this pane in the standalone settings screen. It is thus possible to automatically update these component stacks, or to save changes to them. You may also want to consider creating preference files in the appropriate location on your end user's system (see the `specialFolderPath` function and `query/setRegistry` functions for more information).

Note: Adding additional stacks to the stacks tab is not supported for iOS, Android or HTML5 standalones. Additional stacks may be included via the Copy Files tab.

Including Additional Files

Figure 3 – Standalone Settings – Copy Files

Non-stack files in the application	List other files to be included in the standalone. Use this feature to include help documents, read me files and other resources that you want to include with your standalone each time you build.
Copy Referenced Files	Loops over all image and player objects in stacks and copies any files referenced in the <code>fileName</code> property of these objects into the standalone. Then automatically sets the <code>fileName</code> property to reference these files in the standalone using referenced file paths.
Destination folder	Create a subfolder within your standalone to copy the image and movie files to.

Including Additional Resources

Figure 4 – Standalone Settings – Inclusions

The list of resources available to select for inclusion in a standalone application are a combination of currently installed LiveCode Builder extensions, externals and database drivers (both built-in and those found in user folders), and built-in resources and script libraries.

The following built-in resources are available by default:

Ask Dialog	This option is required if any of your scripts use the "ask" or "ask password" commands. The standalone builder will copy the stack "ask dialog" from the IDE into your standalone as a sub stack. The standalone builder makes a copy of your stack before adding resources to it as part of the build process, so your original stack is unaltered.
Answer Dialog	This option is required if any of your scripts use the "answer" command. Note that this only applies to the dialog form of the command. The answer file / printer / color / effect / folder / page setup / printer and record forms of the command do <i>not</i> require this option. The standalone builder will copy the stack "answer dialog" into your standalone.
Browser	This option is required if your application uses the embedded browser or any <code>revBrowser</code> command.
Browser (CEF)	This option is required if your application uses the embedded CEF browser or any <code>revBrowser</code> command.
Brushes	This option is required if your application uses any of LiveCode's brush cursors. It is not required if your application does not make use of the painting commands. It copies the stack "brushes" into your standalone.
Cursors	This option is required if your application uses any of LiveCode's cursors. It is not required if your application only uses OS cursors. It copies the stack "cursors" into your standalone.
Database	This option is required if your application uses database access or any <code>revDatabase</code> command.
Magnify	This option is required if your application uses the magnify palette.

PDF Printer	This option is required if your application uses the "open printing to pdf" command.
Print Dialog	This option is required if your application uses LiveCode's built-in print or page setup dialogs (e.g. for use on Linux without GTK installed). It is not required if you only display the system printer and page setup dialogs. It copies the stack "print dialog" and "page setup" into your standalone.
SSL & Encryption	This option is required if your application uses any SSL or encryption related commands
XML	This option is required if your application uses any <code>revXML</code> commands

The following script libraries are available by default:

Library Name	Automatic inclusion condition
Animation	This library is unsupported.
DataGrid	DataGrid object.
Geometry	Geometry properties or commands.
Internet	Internet access, including URL, ftp & POST
Printing	<code>revPrintField</code> , <code>revShowPrintDialog</code> and <code>revPrintText</code>
Zip	All <code>revZip</code> commands (but not required for <code>compress/decompress</code>)
Speech	<code>revSpeak</code> and <code>revSpeechVoices</code>
Table	Use of the table object
XMLRPC	Any <code>revXMLRPC</code> commands

The following database drivers are available by default:

- ODBC
- MySQL
- SQLite
- PostgreSQL

Mac Deployment Settings

Figure 5 – Standalone Settings – Mac

Build for Mac OS X 32-bit	Build a standalone that includes a 32 bit slice.
Build for Mac OS X 64-bit	Build a standalone that includes a 64 bit slice.
Application Icon	Choose an application icon to represent the application in the Finder. The icon should be in icns format.
Document Icon	Choose a document icon to represent your application's documents in the Finder. The icon should be in icns format.

Icons for ask / answer dialogs

Choose an icon to display whenever you use the ask or answer commands to display a dialog. On Mac OS X, the convention is that these dialogs should display your application icon. The icon should be stored in your stack as an image, or selected from LiveCode's built-in icons. If you have used a built-in icon, be sure to select the relevant inclusion on the General tab (if you are selecting inclusions manually).

PLIST - enter information and have LiveCode write the PLIST

Have LiveCode fill out the PLIST for your application automatically. The PLIST is a settings file stored in XML format stored as part of every Mac OS X application. It contains information about the application, including its name, version number, copyright notice and document associations. Having LiveCode create this file for you is the recommended option. For more information about PLISTs consult [Apple's developer documentation](#)

Choose a file to import into the application bundle

Choose to import a PLIST file instead of having LiveCode create one. Select this option if you have created your own highly customized PLIST that you want to use for your application in each build you create.

Short version / long version

The version information to be included with your standalone.

Get info string

The visible text displayed in your application's Get Info window by the Finder.

Copyright notice

The copyright notice for your application.

Bundle identifier

A unique identifier for your application used by Mac OS X to identify your application.

Related lessons:

- [Signing and Uploading apps to the Mac App Store](#)

Windows Deployment Settings

Figure 6 – Standalone Settings – Windows

Build for Windows x86

Build a 32-bit standalone for the Microsoft Windows OS.

Build for Windows x86_64

Build a 64-bit standalone for the Microsoft Windows OS.

Application icon

Choose an application icon to represent the application in Windows. The icon should be in .ico format.

Document icon

Choose a document icon to represent your application's documents in Windows. The icon should be in .ico format.

Version information

The version information to be stored as part of your application and displayed in the Windows property inspector and dialogs.

UAC Execution Level Select the user account control level that applies to your application. For more information, consult [MSDN](#)

Linux Deployment Settings

Figure 7 – Standalone Settings – Linux

Build for Linux Build a standalone for 32-bit Linux

Build for Linux x64 Build a standalone for 64-bit Linux

Include Select built-in LiveCode dialogs to include. These dialogs are useful if your application may be run on a system that does not include these dialogs as part of the OS. You do not need to include these dialogs if you are running a recent version of GTK.

iOS Deployment Settings

Figure 8 – Standalone Settings – iOS Basic Settings

Build for iOS iPod, iPhone and iPad Build a standalone iOS iPod, iPhone and iPad

Build for iOS iPod and iPhone Build a standalone iOS iPod and iPhone

Build for iOS iPad Build a standalone iOS iPad

Minimum version Choose the minimum version of iOS that your application should run on

Build 32-bit slice only Only include the 32 bit slice in the standalone instead of both the 32 bit and 64 bit slice

Display Name The name of the application

Version The version number of the application

Beta version Enable Test Flight distribution

Build No. The build number of the application

Internal App ID The application identifier for your application. This should begin with a reverse domain name of a domain you own.

Profile The provisioning profile to sign the application with.

iPhone Status Bar Choose whether to show or hide the status bar while the application is running on iPhone

iPad Status Bar Choose whether to show or hide the status bar while the application is running on iPad

Status Bar Style	Set the style of the status bar while the application is running
iPhone Initial Orientation	Set the orientation on iPhone of the application on the screen when it initially starts
iPad Supported Initial Orientation	Set the possible orientations on iPad of the application on the screen when it initially starts
Custom URL Scheme	Add a custom url scheme to the application so that it will launch when a url is opened that uses the scheme
Mapping file	Set a font name mapping file which should contain lines of <code><mapped name>=<PostScript name></code>
Telephony	Choose whether telephony is required, prohibited or not applicable in order for the application to function
Peer-Peer	Choose whether peer to peer is required, prohibited or not applicable in order for the application to function
SMS	Choose whether SMS is required, prohibited or not applicable in order for the application to function
Still Camera	Choose whether a still camera is required, prohibited or not applicable in order for the application to function
Auto-focus Camera	Choose whether an auto-focus camera is required, prohibited or not applicable in order for the application to function
Front-facing Camera	Choose whether a front-facing camera is required, prohibited or not applicable in order for the application to function
Accelerometer	Choose whether an accelerometer is required, prohibited or not applicable in order for the application to function
Location Services	Choose whether location services is required, prohibited or not applicable in order for the application to function
GPS	Choose whether GPS is required, prohibited or not applicable in order for the application to function
Magnetometer	Choose whether a magnetometer is required, prohibited or not applicable in order for the application to function
Microphone	Choose whether a microphone is required, prohibited or not applicable in order for the application to function
Game-Kit	Choose whether Game-Kit is required, prohibited or not applicable in order for the application to function
WiFi	Choose whether WiFi is required, prohibited or not applicable in order for the application to function
Magnetometer	Choose whether a magnetometer is required, prohibited or not applicable in order for the application to function
OpenGL ES 1.1	Choose whether OpenGL ES 1.1 is required, prohibited or not applicable in order for the application to function
OpenGL ES 2.0	Choose whether OpenGL ES 2.0 is required, prohibited or not applicable in order for the application to function

Figure 9 – Standalone Settings – iOS Requirements

Location Authorization Type	Choose whether to authorize location services for just when the application is in use or for background use also
------------------------------------	--

Persistent WiFi	Indicate the application requires persistent WiFi in order to function
File Sharing	Indicate the application uses file sharing
Local Notifications	Indicate the application sends local notifications
Push Notifications	Indicate the application sends push notifications
Disable ATS	Disabling ATS allows your application to load insecure websites (not recommended).
Enable Background Execution	By default LiveCode applications will exit when the user suspends them. This option allows the applications to remain open while in the background.
Background Audio	Enable if the application plays audio while in the background.
Location Update	Enable if the application receives location updates while in the background.
VoIP	Enable if the application handles VoIP calls while in the background.
Newsstand Downloads	Enable if the application receives newsstand downloads while in the background.
External Accessory Communication	Enable if the application communicates with external accessories while in the background.
Use Bluetooth Low Energy Accessories	Enable if the application communicates with bluetooth low energy accessories while in the background.
Acts as Bluetooth Low Energy Accessory	Enable if the application acts as a bluetooth low energy accessory while in the background.
Background Fetch	Enable if the application fetches data while in the background.
Remote Notifications	Enable if the application handles remote notifications while in the background.
App URL Query Whitelist	Specify a list of url schemes the application needs to query to determine if there is an application on the device capable of handling the scheme.

Figure 10 – Standalone Settings – iOS Icons

Prerendered Icon	Indicate the icons are prerendered for display
AppStore	Choose a 1024x1024 pixel png image
iPhone	Choose a 57x57 pixel png image
Hi-Res iPhone	Choose a 114x114 pixel png image
iOS 7 Hi-Res iPhone	Choose a 120x120 pixel png image
iPhone 6 Plus	Choose a 160x160 pixel png image
iPhone X	Choose a 180x180 pixel png image
iPad	Choose a 72x72 pixel png image
Hi-Res iPad	Choose a 144x144 pixel png image
iOS 7 iPad	Choose a 76x76 pixel png image
iOS 7 Hi-Res iPad	Choose a 152x152 pixel png image
iPad Pro 12.9	Choose a 167x167 pixel png image

Figure 11 – Standalone Settings – iOS Splash

Launch Image	Choose a png image. This will be centered on screen.
2x Launch Image	Choose a png image. This will be centered on screen.
3x Launch Image	Choose a png image. This will be centered on screen.
Dark Mode Launch Image	Choose a png image for use in dark mode. This will be centered on screen.
Dark Mode 2x Launch Image	Choose a png image for use in dark mode. This will be centered on screen.
Dark Mode 3x Launch Image	Choose a png image for use in dark mode. This will be centered on screen.
Background Color	Choose a background color for transparent areas and/or areas of the screen the launch image does not cover.
Use system background color	Use the system background color instead of a chosen color. This will use a dark color in dark mode and light color in light mode.

Related lessons:

- [How do I Become an iOS Developer?](#)
- [How do I build an iOS application?](#)

Android Deployment Settings

Figure 12 – Standalone Settings – Android

Build for Android armv7	Include armv7 binaries in the Android standalone
Build for Android arm64	Include arm64 binaries in the Android standalone
Build for Android x86	Include x86 binaries in the Android standalone
Build for Android x86_64	Include x86_64 binaries in the Android standalone
Label	The application name
Identifier	The application identifier for your application. This should begin with a reverse domain name of a domain you own.
Version Name	The version number of your application
Version Code	The build number of your application
Icon	The application icon
Splash	The application splash screen (applicable to the Educational Personal license only)
Signing	Choose to sign for development, with your own key or not to sign.
Key	Choose a Java keystore file to sign with.
Install Location	Allow installation onto external storage.
Custom URL Scheme	Add a custom url scheme to the application so that it will launch when a url is opened that uses the scheme.

Push Sender ID	The project number from Google's Cloud Messaging API. See the How do I use Push Notifications with Android? lesson.
Status Bar Icon	The icon shown in the status bar for a notification
Hardware Accelerated	Enable hardware acceleration of the application
Initial Orientation	The orientation on the device when the application initially launches
Status Bar	Set the visibility of the status bar
In App Purchasing	Specify the store used for the in-app purchasing API
Minimum Android Version	Choose the minimum version of Android the application should run on
Camera	Choose whether a camera is required, prohibited or not applicable in order for the application to function
Camera Autofocus	Choose whether camera autofocus is required, prohibited or not applicable in order for the application to function
Camera Flash	Choose whether a camera flash is required, prohibited or not applicable in order for the application to function
Front Camera	Choose whether a front camera is required, prohibited or not applicable in order for the application to function
Accelerometer	Choose whether an accelerometer is required, prohibited or not applicable in order for the application to function
Telephony	Choose whether telephony is required, prohibited or not applicable in order for the application to function
Telephony CDMA	Choose whether CDMA telephony is required, prohibited or not applicable in order for the application to function
Telephony GSM	Choose whether GSM telephony is required, prohibited or not applicable in order for the application to function
Fake Touch	Choose whether fake touch is required, prohibited or not applicable in order for the application to function
Touchscreen	Choose whether a touchscreen is required, prohibited or not applicable in order for the application to function
Multitouch	Choose whether multitouch is required, prohibited or not applicable in order for the application to function
Multitouch Distinct	Choose whether multitouch distinct is required, prohibited or not applicable in order for the application to function
Multitouch Jazzhand	Choose whether multitouch jazzhand is required, prohibited or not applicable in order for the application to function
Write External Storage	Request permission to write to external storage
Internet	Request permission to for internet access
Camera	Request permission to access the camera
Read Contacts	Request permission to read the user's contact data
Write Contacts	Request permission to write to the user's contact data
NFC Tag Reading	Request permission to read NFC tags
Fine Location	Request permission to access the device's fine location
Course Location	Request permission to access the device's course location
Vibration	Request permission to vibrate the device
Idle Timer	Request permission for the idle timer
Ad Support	Request permission for ad support

Related lessons:

- [LiveCode and Android Studio](#)
- [The Basics: How do I Create Hello World on Android?](#)

HTML5 Deployment Settings

Figure 13 – Standalone Settings – HTML5

Build for HTML5 Build a standalone for HTML5

Almost every Internet-connected device has a web browser. If your application can run in a browser, your app can be used anywhere and by anyone, without any need to download or install it.

Related lessons:

- [How Do I Put My First App On the Web](#)

Supported browsers

The following browsers are supported:

- [Mozilla Firefox](#) 40.0 (or newer)
- [Google Chrome](#) 44 (or newer)
- [Safari for Mac](#) 9.0 (or newer)

HTML5 engine features

The HTML5 engine in this release of LiveCode has a limited range of features. You can:

- deploy single or multiple stack applications with embedded resources. Stacks other than the main stack will open in their own floating container windows.
- use most of the engine's built-in controls and graphics capabilities.
- read and write temporary files in a special virtual filesystem (which is erased when the user navigates away from the page)
- use LiveCode Builder widgets and extensions
- interact with JavaScript code in the web page using `do <script> as "JavaScript"`
- call JavaScript from LiveCode Builder widgets and extensions using the `com.livecode.emscripten` module
- implement widgets by embedding DOM elements as native layers
- perform basic networking operations using the **load** command

Several important features are not yet supported:

- multimedia (the "player" control)

Two important unsupported features are unlikely to be added in the near future:

- operations that need to pause the script while something happens (e.g. `wait 10`)
- externals (including `revdb`)

Contents of the HTML5 standalone

The HTML5 standalone contains four files:

- A standalone archive, named `standalone.zip` by default. This file contains your application and all of the resources that it depends on. When the engine runs, the filesystem that's visible to the engine (e.g. via the `open file` syntax) is based on the contents of the standalone archive.
- The engine itself, which consists of two files. The `.js` file contains the engine's executable code, and the `.html.mem` file contains essential data that's needed for the engine to run. These files are always the same, and only change when LiveCode is upgraded.
- A test HTML page. This can be opened in a browser and will correctly prepare, download and start your HTML5 app in a convenient test environment.

Advanced: HTML5 standalone filesystem

JavaScript applications running in a browser don't have access to the host system's filesystem. Instead, the filesystem-related features of LiveCode, such as `open file`, use a virtual filesystem (VFS) that exists only in memory. This filesystem is initialised before the engine starts, and is reset and its content discarded when the engine stops (when the user closes the browser view or navigates to a different page).

During engine startup, the VFS is populated from the contents of the `standalone.zip` file that's created by the HTML5 deployment process. All of the initial files are stored in `/boot/` in the VFS.

There are several special files & directories in the `/boot/` directory:

- `/boot/___startup.data`: the standalone data including the mainstack, extensions and auxiliary stackfiles and startup script.
- `/boot/fonts/basefont.ttf`: the font used by the engine
- `/boot/standalone/`: the `defaultFolder` when the engine starts, and the location where additional assets selected using the "Copy files" page of the standalone builder are placed

In general, if you wish to add new files or directories to the `standalone.zip` archive, it is best to add them outside the `/boot/` directory tree.

Advanced: Embedding an HTML5 standalone in a web page

The default HTML5 page provided by the HTML5 standalone builder is designed for testing and debugging purposes. However, you may want to embed the standalone engine in a more visually appealing page. To do this, you require three elements: 1) a canvas, 2) a JavaScript `Module` object, and 3) an HTML `<script>` element that downloads the engine.

The canvas

The engine renders into a HTML5 `<canvas>` element contained within a `<div>` element. There are some important considerations when creating the canvas & div:

- both the canvas and div must have absolutely no border, or mouse coordinate calculations will be incorrect
- they will be automatically resized by the engine to match the size of your stack, so don't attempt to set their size using HTML or CSS
- the canvas should be the only element within the containing div, which may be used to hold additional elements as native layers are added to the app.

- the canvas needs to be easily uniquely identifiable, so that the engine can find it.

The absolute minimum canvas element would look something like this:

```
<div><canvas style="border: 0px none;" id="canvas"
oncontextmenu="event.preventDefault();"></canvas></div>
```

By default, most web browsers will indicate when the canvas has focus by displaying a highlighted outline. This helps users identify which part of the web page is capturing their key presses. You can usually disable this outline by adding `outline: none;` to the canvas's CSS styles.

The Module object

The top-level JavaScript `Module` object contains the parameters that control how the engine runs. At minimum, you need only specify the `Module.canvas`, which should be your canvas element.

The absolute minimum `Module` object declaration would look something like:

```
<script type="text/javascript">
var Module = {
  canvas: document.getElementById('canvas'),
};
</script>
```

Engine download

The engine is quite a large JavaScript file, so it's downloaded asynchronously in order to let the rest of the page finish loading and start being displayed.

Quite straightforwardly:

```
<script async type="text/javascript" src="standalone-<version>.js"></script>
```

Make sure to replace `<version>` as appropriate.

Bringing it all together

Here's the complete skeleton web page for an HTML5 standalone:

```
<html>
  <body>
    <div>
      <canvas style="border: 0px none;" id="canvas"
oncontextmenu="event.preventDefault()"></canvas>
    </div>

    <script type="text/javascript">
      var Module = { canvas: document.getElementById('canvas') };
    </script>
    <script async type="text/javascript" src="standalone-community.js"></script>
  </body>
</html>
```

Advanced: Speeding up engine download

Currently, the engine files are almost 30 MB, which is a lot to download before the engine can start. It is possible to speed up the download by enabling deflate compression in the web server configuration.

Enabling deflate compression reduces the total download size to around 6.3 MB. It's recommended to pre-compress the engine with `gzip`, and then configure your web server to serve the pre-compressed files.

- For the Apache web server, configure `mod_deflate` to serve [pre-compressed content](#)
- For the NGINX web server, add `gzip_static on;` to your configuration.

Advanced: Customizing the Module object

There are a number of LiveCode-specific `Module` attributes that you can modify to affect how the engine behaves:

- `Module.livecodeStandalone`: the filename of the standalone archive (default `standalone.zip`)
- `Module.livecodeStandalonePrefixURL`: Prepended to the standalone archive filename to construct its full URL (default empty)
- `Module.livecodeStandaloneRequest`: If you assign a network request to this attribute (before the engine runs), then it will use that request for the standalone archive instead of automatically starting a download for you. This means that you can, in your HTML, fire off a request for the standalone before the engine script actually arrives. For this to work, the network request should be an `XMLHttpRequest` with its `responseType` set to `arraybuffer`.

See also Emscripten's [Module object documentation](#).

Bug Reports

Figure 14 – Standalone Settings – Bug Reports

Include Error Reporting Dialog	Include an error reporting stack in your standalone. You should select this option if you are testing your application and want details of any errors in your standalone, or if you have not included your own error reporting routines in your stacks.
htmlText for dialog	The text to display to the user in the dialog that comes up when an error is encountered. This text should be in LiveCode-compatible HTML format. Create and format the text in a LiveCode field then copy the field's HTMLText property.
Dialog icon	The icon to display in the error dialog. This should be stored as an image in your stack.
Allow user to enter comments	Display a box for the user to give you more information. This information will be included in the report.
Allow user to save report to file	Allow the user to save the report to a file. Select this option if you want users to save an error report and send it to you.
Allow user to email report	Allow the user to email the report. Select this option if you want the user to be able to send you or your technical support department details of the error. This option loads up the system default email client and populates the email with the contents of the error report and the user's comments. The To: field is sent to the email address specified in the email address field.

Testing Your Application

Testing an app is straightforward:

- 1) Open your stack in the LiveCode IDE
- 2) Select **File → Standalone Application Settings...** from the menu bar
- 3) Select the settings for your app
- 4) Make sure that checkboxes for each platform you wish to build for are checked as the chosen platforms will govern which test targets are available.
- 5) Close the standalone settings window
- 6) Choose **Development → Test Target → Your Target** from the menu bar to select the target to test deployment to
- 7) Click the **Test** button or choose **Development → Test** from the menu bar
- 8) The standalone will be built and deployed to the target and launched. If you have the pro features pack, then the remote debugger will detect any execution errors and present them.

Note: iOS devices connected via USB are detected as test targets when you have the pro features pack and apps will be installed on them, however, they will not be automatically launched.

Extending LiveCode

Introduction

LiveCode has a simple way to extend the functionality or control set of your app using separately compiled modules. This guide will take you through the process of installing and using these new widgets and libraries and for the adventurous among you, provide a guide to building and sharing your own extensions.

LiveCode Builder Extensions

To make it possible to create extensions and plug them into the LiveCode engine we've created a new flavor of our language called **LiveCode Builder**. LiveCode Builder looks a lot like LiveCode Script so should feel familiar for any seasoned LiveCode developer. There is lots of new syntax which exposes parts of the LiveCode engine that were only previously available to those who were skilled c/c++ developers.

To learn more about LiveCode Builder and creating extensions, read on.

Warning: It is important to stress right at the start that **no aspect of LiveCode Builder should be considered final. Every piece of syntax in LiveCode Builder is subject to change.**

Creating LiveCode Builder Extensions

The IDE provides an "Extension Builder" tool to help make development and packaging of extensions as easy as possible.

Note: LiveCode Builder is a different flavor of LiveCode so it is not possible to edit LiveCode Builder scripts in the main LiveCode Script IDE.

Extension Builder

Open the "Extension Builder" from the tools menu:

Tools > Extension Builder

-
1. Shows the currently selected extension.
 2. Data that the builder was able to parse from the directory such as icons, resources, API's the user guides.
 3. Console: Shows status, error and log messages.
 4. Test: Creates a stack, compiles the extensions and creates an instance.
 5. Script: Opens the lcb script in an external default editor.
 6. Install: Installs the extension into the IDE
 7. Uninstall: Uninstalls the extension from the IDE
 8. Package: Creates a .lce package which can uploaded to the extension store. It is placed in the extension directory

Note: A great way to get started is to tweak the script of one of our widget examples.

Using the selection dropdown (1), you can select the extension you wish to develop or click the "open" icon in the header back to locate an extension you've not loaded before.

Create your own simple widget

The main difference between widgets and libraries is that a widget draws to a pre-existing canvas. Apart from that, the authoring process is much the same for both extension types.

Create a .lcb file

We recommend using the Atom text editor, available at <https://atom.io/>. A LiveCode package is available which provides some colorization as well as indentation. If you prefer to use TextWrangler, there is a colorizing script [here](#). It should be placed in /Application Support/TextWrangler/Language Modules/

Start by creating a plain text file in a new directory and save it to disk with the extension ".lcb":

```
/Desktop/widgettest/main.lcb
```

Note: The extension builder currently relies on there being only one main module file in a given directory.

Declare Type and Identifier

Start by declaring the type of extension, either "widget" or "library" followed by your identifier (See "Select A Developer ID" below).

```
widget community.livecode.beaumont.pinkCircle
## Code for your widget
end widget
```

This is the unique identifier by which your extension will be referred to by the LiveCode Engine.

Note: All LiveCode authored widgets are in the application package at /Tools/Extensions/. To play with one of these extensions we recommend copying the folder from the application package to your desktop and loading it from there. Changing the extension identifier in the source code will ensure it doesn't conflict with the existing extension.

Declare Metadata

Next, provide metadata to help LiveCode display your widget correctly in the IDE and in the online portal.

```
widget community.livecode.beaumont.pinkCircle

metadata title is "My Pink Circle"
metadata author is "Benjamin Beaumont"
metadata version is "1.0.0"
metadata platforms is "desktop,mobile"

end widget
```

Note: If the module makes use of external code that is only available on specific Operating

Systems or Platforms, use the "os" and/or "platforms" metadata keys. The values are the same as can be found in the LiveCode Documentation Format Reference. This data will also appear in the dictionary.

Importing libraries

The LiveCode builder syntax is broken down into **modules**. There are 3 classes of module:

Type	Description
Default	These modules are part of LiveCode builder and are included by default. Their syntax is always available to you as a LiveCode developer.
Optional	These modules are created and distributed by LiveCode Ltd and must be imported by the extension developer in order to make use of their syntax.
Custom	These modules are created and distributed through the online portal and must be imported by the extension developer in order to make use of their syntax.

LiveCode builder contains the following modules:

Module	Type	Description
com.livecode.canvas	Optional	Provides the syntax and types for 2D drawing allowing developers to draw to a canvas. Required if creating a widget.
com.livecode.widget	Optional	Contains syntax specific to widget building such as "my width" and "the mouse position".
com.livecode.engine	Optional	Contains syntax for all extension building such as "dispatch" and "log".
com.livecode.arithmetic	Default	Contains syntax for basic mathematical operations.
com.livecode.array	Default	Contains syntax for operations on arrays.
com.livecode.assert	Default	Contains syntax for making assertions about program state.
com.livecode.binary	Default	Contains syntax for operations on binary data.
com.livecode.bitwise	Default	Contains syntax for bitwise logical operators.
com.livecode.byte	Default	Contains syntax for operations on byte chunks.
com.livecode.char	Default	Contains syntax for operations on char chunks.
com.livecode.codeunit	Default	Contains syntax for operations on codeunit chunks.
com.livecode.date	Default	Contains syntax for accessing the date and time.
com.livecode.file	Default	Contains syntax for file I/O operations.
com.livecode.foreign	Default	Provides the type bindings for foreign types.
com.livecode.java	Default	Provides helper functions and types for interfacing with Java.
com.livecode.list	Default	Contains syntax for operations on lists.
com.livecode.logic	Default	Contains syntax for logical operators.
com.livecode.math	Default	Contains syntax for mathematical operations.
com.livecode.mathfoundation	Default	Contains syntax for foundational mathematical operations.
com.livecode.objc	Default	Provides helper functions and types for interfacing with Objective-C.
com.livecode.sort	Default	Contains syntax for sorting operations.
com.livecode.stream	Default	Contains syntax for stream I/O operations.
com.livecode.string	Default	Contains syntax for operations on strings.
com.livecode.system	Default	Contains syntax for accessing system information.
com.livecode.type	Default	Contains syntax for general operations on types.

Module	Type	Description
com.livecode.typeconvert	Default	Contains syntax for conversions between types.
com.livecode.unittest	Default	Contains syntax for unit testing LiveCode Builder programs.

Warning! Module names are subject to change.

The LiveCode dictionary has a full list of all available syntax as well as the module each belongs to. As a general rule we recommend importing all three optional modules whenever developing widgets.

```

widget community.livecode.beaumont.pinkCircle

use com.livecode.canvas
use com.livecode.widget
use com.livecode.engine

metadata title is "My Pink Circle"
metadata author is "Benjamin Beaumont"
metadata version is "1.0.0"

end widget

```

Core Handlers

There are several core handlers that any widget developer should implement:

Handler	Description
OnPaint	The <i>OnPaint</i> message is sent to your widget whenever LiveCode requires it to redraw. The performance of your widget is tied primarily to this handler and should be kept as efficient as possible.
OnCreate	The <i>OnCreate</i> message is sent to your widget when it is first created by LiveCode. This can be used to initialise default data and where applicable, reduce the burden for calculating constants etc in the onPaint handler.
OnGeometryChanged	The <i>OnGeometryChanged</i> message is sent when the control is changed in size.
OnSave	The <i>OnSave</i> message is sent when your widget is about to be destroyed and enables the widget to save data set on the widget.
OnLoad	The <i>OnLoad</i> message is sent when your widget is created and enables the widget to retrieve data saved on the widget.

In the first instance we are going to create a widget with no settable properties, just using the `OnPaint` handler.

```

widget community.livecode.beaumont.pinkCircle

metadata title is "My Pink Circle"
metadata author is "Benjamin Beaumont"
metadata version is "1.0.0"

public handler OnPaint()
    // Draw widget
end handler

end widget

```

Draw a Pink Circle

```

widget community.livecode.beaumont.pinkCircle

metadata title is "My Pink Circle"
metadata author is "Benjamin Beaumont"
metadata version is "1.0.0"

use com.livecode.canvas

public handler OnPaint()
    // Create a path with a radius of half the width of the canvas
    variable tCirclePath as Path
    put circle path centered at point [my width / 2, my height / 2] with radius (my
width/2) into tCirclePath

    // Set the paint that will be used to fill the circle to a solid
    // pink color
    set the paint of this canvas to solid paint with color [1, 0, 1]

    // Fill the path
    fill tCirclePath on this canvas
end handler

end widget

```

Test the Code

Now open the extension builder stack as shown above and click on the folder icon at the top right to load your **.lcb** file into the builder.

Click test. Your widget should be displayed on the new stack. If you can't see it, check behind the extension stack.

Properties

In order to make a widget useful to end users it is likely that you'll want to expose properties that allow them to specify how your widget should behave.

To specify a property you must provide a name and the method to get and set the property.

```
property <name> get <variable/handler> set <variable/handler>
```

The simplest properties to get/set are numbers or strings. So lets create a `circleMargin` property that allows users to define a margin.

```
property circleMargin get mMargin set setMargin
```

In the above example, when the `circleMargin` property is requested, the variable `mMargin` is returned; when the property is set, the handler `setMargin` is called. To have a property linked directly to the value of a variable, simply provide the variable name. There will be no other side effects - notably, a redraw will not automatically be triggered in the case that a variable name is used for a setter. To process the value coming from and going into LiveCode Script, or to add side-effects when getting and setting properties, provide handler names. In our case we're defining a setter for the `circleMargin` property because we need to trigger a redraw when it is set.

```
private variable mMargin as Real
```

We'll define a member variable to store the value for the margin. LiveCode Builder is typed so you must also specify the type of your variable. Remember, the canvas you are drawing to has subpixel precision so our margin is a real number rather than an integer. For a full list of types available in LiveCode Builder please see the Typing section of the language specification guide.

We also suggest a naming convention for variables in the section on variable and case sensitivity.

Finally we have to implement our `setMargin` handler.

```
public handler setMargin(in pMargin as Real) returns nothing
    put pMargin into mMargin
    redraw all
end handler
```

Implementing the "setter" ourselves provides us with a little more flexibility. In this case when the property is set we want our pink circle to immediately redraw to reflect the property change. We do this by calling "redraw all".

To test the property click "test" and from the message box set the property.

```
set the circleMargin of widget 1 to 15
```

Full Example

```

widget community.livecode.beaumont.pinkCircle

metadata title is "My Pink Circle"
metadata author is "Benjamin Beaumont"
metadata version is "1.0.0"

use com.livecode.canvas
use com.livecode.widget
use com.livecode.engine

// Properties
property circleMargin get mMargin set setMargin

// Local variables
private variable mMargin as Real

public handler onCreate()
    put 0 into mMargin
end handler

public handler OnPaint()
    // Create a path with a radius of half the width of the canvas
    variable tCirclePath as Path
    put circle path centered at point [my width / 2, my height / 2] with radius (my
width/2) into tCirclePath

    // Set the paint that will be used to fill the circle to a solid
    // pink color
    set the paint of this canvas to solid paint with color [1, 0, 1]

    // Fill the path
    fill tCirclePath on this canvas
end handler

public handler setMargin(in pMargin as Real)
    put pMargin into mMargin
    redraw all
end handler

end widget

```

Loading and Saving widget data

When a stack containing a widget is saved, the widget's *OnSave* handler is called by the engine. The *OnSave* handler has an array out parameter - you can fill this array with whatever widget data is needed to reconstruct the widget. LiveCode saves this data along with instances of the widget in the stack file.

```

public handler OnSave(out rProperties as Array)
    put the empty array into rProperties

    put mMargin into rProperties["margin"]

    return rProperties
end handler

```

This same array is passed as a parameter to an *OnLoad* handler which is called when the stack the widget is on is loaded.

```

public handler OnLoad(in pProperties as Array)
    put pProperties["margin"] into mMargin
end handler

```

Understanding Error Messages

Clicking on the "test" button causes the extension builder to compile your source code file (**.lcb**) and produce a compiled module file (**.lcm**). If an error is encountered it is output in the "console" section of the builder:

```
Error: <source path>: <line number>: <char number>: <error message>
```

- *source path* - The path the .lcb file that is being compiled
- *line number* - The line number in the script on which the error occurred.
- *char number* - The character number on which the error occurred.
- *error message* - The description of the error.

LiveCode Builder API

To start creating more complex widgets see the LiveCode Builder api in the documentation stack in the IDE.

-
1. Open the dictionary (and stay in the 'API' tab)
 2. Click the drop-down button
 3. Choose the "LiveCode Builder" API
 4. The list of all the syntax available

Hello World library

A extension library is written in much the same way as the above widget. The only difference is that all the public handlers implemented are inserted into the message path when the extension is loaded.

For example, here is a simple library with one handler which simply returns the string "Hello World!":

```

library community.livecode.elanorb.helloworld

metadata title is "Hello World Library"
metadata author is "Elanor Buchanan"
metadata version is "1.0.0"

public handler sayHello()
  return "Hello World!"
end handler

end library

```

Once this is compiled and loaded, the sayHello handler would be in the message path, before the backscripts. LiveCode Builder library handlers can be invoked in both command and function form:

```

sayHello
put the result -- outputs "Hello World!"

put sayHello() -- outputs "Hello World!"

```

Loading and Creating Widgets Manually

To load a compiled module file call:

```
load extension <filepath.lcm>
```

To create a widget from script call:

```
create widget <name> as <identifier>
```

e.g.

```
create widget "myWidget" as "com.livecode.extensions.waddingham.clock"
```

Extensions Course

See the online [extensions course](#) for a thorough introduction to LiveCode Builder, complete with examples. The following extension examples are worked through:

- Hello World library
- Extended Hello World library
- Rotated Text widget
- Extended Rotated Text widget
- Pie Chart widget
- Modifying the Line Graph widget

Canvas API

If you are drawing to the widget canvas in the `OnPaint` handler (as opposed to [wrapping native views](#)) it is a good idea to familiarise yourself with the canvas API. It provides a set of operations for creating, measuring, transforming and drawing paths and text.

Using path objects

Paths corresponding to standard shapes can be created using the various path 'constructors':

- Line: `line` path from `mFrom` to `mTo`
- Rectangle: `rectangle` path of `mRect`
- Circle: `circle` path centered at `mCenter` with radius `mRadius`
- Ellips: `ellipse` path centered at `mPoint` with radii `mRadii`
- Rounded Rectangle:
`rounded rectangle` path of `mRect` with (radius `mRadius` | radii `mRadii`)
- Arc:
`arc` path centered at `mCenter` with [radius `mRadius` | radii `mRadii`] from `mStartAngle` to `mEndAngle`
- Sector:
`sector` path centered at `mCenter` with [radius `mRadius` | radii `mRadii`] from `mStartAngle` to `mEndAngle`
- Segment:
`segment` path centered at `mCenter` with [radius `mRadius` | radii `mRadii`] from `mStartAngle` to `mEndAngle`
- List of points: (`polygon` | `polyline`) path with points `mPoints`

To create a path using the path operations, start with an empty path object:

```
// Create a new empty path
variable tPath as Path
put the empty path into tPath
```

Then use the canvas path syntax to build the path. Simple paths with no subpaths (i.e. ones you can draw without lifting the pencil), can be created using the following operations:

- Line: `line` to `mPoint` on `mPath`
- Arc: `arc` through `mThrough` to `mTo` with radius `mRadius` on `mPath`
- Curve: `curve` through `mThroughA` [then `mThroughB`] to `mTo` on `mPath`
- Elliptic arc:
`arc` to `mEnd` with radii `mRadii` rotated by `mAngle` [taking (largest | smallest) (clockwise | anticlockwise) route] on `mPath`
- Close path: `close` path on `mPath`

Paths made up of disjoint subpaths (i.e. ones you need to lift the pencil for) can be created using the move operation:

- Move: `move` to `mPoint` on `mPath`

For example, to create a path consisting of a pair of parallel lines:

```
public handler OnPaint()  
  // Create a new empty path  
  variable tPath as Path  
  put the empty path into tPath  
  
  // Begin a new subpath of tPath  
  move to point [50, 50] on tPath  
  
  // Add a line to tPath  
  line to point [100, 50] on tPath  
  
  // Begin a new subpath of tPath  
  move to point [50, 100] on tPath  
  
  // Add a line to tPath  
  line to point [100, 100] on tPath  
  
  stroke tPath on this canvas  
end handler
```

The following example illustrates how the parameters to elliptic arcs work. Take the ellipse with horizontal radius 50 and vertical radius 25. The red and green arcs start from the same point; the red is the smallest clockwise route to a given point, the green is the largest anticlockwise route. Together they make up the whole ellipse:

```

public handler OnPaint()
    variable tSmallest as Path
    put the empty path into tSmallest

    // Start at [50, 50]
    move to point [50, 50] on tSmallest

    // Continue path with an arc to 100, 25
    arc to point [100, 25] with radii [50, 25] rotated by 0 taking smallest
    clockwise route on tSmallest

    variable tLargest as Path
    put the empty path into tLargest

    // Start at [50, 50]
    move to point [50, 50] on tLargest

    // Continue path with an arc to 100, 25
    arc to point [100, 25] with radii [50, 25] rotated by 0 taking largest
    anticlockwise route on tLargest

    set the paint of this canvas to solid paint with color [255,0,0]
    stroke tSmallest on this canvas

    set the paint of this canvas to solid paint with color [0,255,0]
    stroke tLargest on this canvas
end handler

```

Paths may also be created using SVG instructions, for example:

```

public handler OnPaint()
    variable tPath as Path
    put path "M10,10 L50,100 Q100,100 100,50 C75,50 50,25 50,10z" into tPath
    stroke tPath on this canvas
end handler

```

Produces the following:

Rendering text

There is essentially only one piece of syntax for rendering text:

```
fill text mText at (mPoint | mAlignment of mRect) on mCanvas
```

for example, a widget that simply displays its name at its center would have the following `OnPaint` handler:

```
public handler OnPaint()
    fill text my name at center of my bounds on this canvas
end handler
```

Text can be measured using the

```
measure mText on mCanvas
```

syntax. This can be used for example to do text wrapping, by calculating the widths of successive chunks of text and inserting line breaks where appropriate.

Transformations

Both Path objects and canvases can be transformed using the standard affine transforms - either using the specific translate, rotate or scale syntax, or arbitrary transform matrices.

Composed Widgets

Widgets can either be 'host' widgets, as in the previous example, created when a widget is directly embedded in a stack, or 'child' widgets which are created when a widget is used as a child widget within another widget. The syntax for composed widgets is included in the `com.livecode.widget` module.

A simple composed widget

This composed widget example composes the clock widget and the selector widget, to create a version of the clock widget with adjustable time zone.

The label at the top of the widget reflects which portion of the widget the mouse is over.


```

widget com.livecode.extensions.example.simplecomposed

use com.livecode.canvas
use com.livecode.widget

metadata title is "Simple Composed Widget"
metadata author is "LiveCode"
metadata version is "1.0.0"

private variable mInsideChild as String
private variable mInside as Boolean

private variable mSelector as Widget
private variable mClock as Widget

public handler OnCreate()
  put false into mInside
  put the empty string into mInsideChild

  put a new widget "com.livecode.extensions.livecode.selector" into mSelector
  set property "numSelections" of mSelector to 6
  set annotation "Name" of mSelector to "Selector"

  put a new widget "com.livecode.extensions.livecode.clock" into mClock
  set annotation "Name" of mClock to "Clock"

  place mSelector
  place mClock
end handler

end widget

```

Notice that Widget is a variable type. This widget stores references to its child widgets in private variables. In the `OnCreate` handler, the widget objects are created, stored in the private variables and 'placed'. Child widgets can be stored as variables and manipulated without actually being drawn to a canvas if they are unplaced. Properties implemented by child widgets can be got and set using the `property <propName> of <child widget>` syntax. Placing a widget ensures that they are drawn, in placement order. Setting an annotation of a child widget assigns it a tag so that when an unknown child widget is returned by an operator, its annotation can be used to identify it.

```

public handler OnMouseEnter()
    put true into mInside

    if the target is not nothing then
        put annotation "Name" of the target into mInsideChild
    end if

    redraw all
end handler

public handler OnMouseLeave()
    if the target is not nothing then
        put the empty string into mInsideChild
    end if

    put false into mInside

    redraw all
end handler

```

In the *OnMouseEnter* and *OnMouseLeave* handlers, the *target* is used to obtain a reference to the child widget that triggered the *OnMouseEnter* and *OnMouseLeave* events, and the previously assigned annotation put into the `mInsideChild` variable, which in turn is rendered to the canvas in the *OnPaint* handler.

```

public handler OnPaint()
    set the paint of this canvas to solid paint with color [0.75, 0.75, 0.75]
    fill rectangle path of my bounds on this canvas

    if mInside then
        set the paint of this canvas to solid paint with color [1.0, 0.0, 0.0]
        set the stroke width of this canvas to 4.0
        stroke rectangle path of my bounds on this canvas
    end if

    if mInsideChild is not the empty string then
        set the paint of this canvas to solid paint with color [0.0, 0.0, 0.0]
        fill text mInsideChild at top of my bounds on this canvas
    end if
end handler

```

The rectangle of each child widget is controlled using the following syntax:

- the rectangle of <widget> - Enables manipulation of the rectangle property of a child widget.
- the width of <widget> - Enables manipulation of the width property of a child widget.
- the height of <widget> - Enables manipulation of the height property of a child widget.
- the location of <widget> - Enables manipulation of the location property of a child widget.

```
public handler OnGeometryChanged()
```

```
set the rectangle of mSelector to rectangle [ 0, 20, my width, 50 ]
set the rectangle of mClock to rectangle [ 0, 50, my width, my height ]
```

end handler

Finally, messages posted by child widgets can be handled in the direct parent by handling the appropriate message (prepending 'On'). For example, the selector widget posts *optionChanged* when one of its numbers is selected. This is handled by this composed widget example in an *OnOptionChanged* handler.

```
public handler OnOptionChanged(in pIndex)
    set property "timeZone" of mClock to pIndex - 1
end handler
```

Using native APIs for libraries

One of the most powerful features of LiveCode Builder is the foreign function interface (FFI). It can be used to access native APIs on all supported platforms.

The fundamental concept involved in using the FFI is the foreign handler binding. This is a string containing information about the native functionality that is being bound to.

The format of the binding string varies slightly depending on the language of the foreign handler - for a complete specification, see the LiveCode Builder Language Reference.

C

In order to bind to C functions from a dynamic library, there is one essential thing to know: the signature of the function.

Note: On Windows, it is also necessary to find out the calling convention of the function. In the majority of cases if binding to functions in the Windows API, the calling convention will be `stdcall`.

The declared types of the foreign handler must match the signature, so that the FFI call can be computed correctly. For example, to bind to the `atof` C standard library function that converts a C-string to a double,

```
double atof(const char *str)
```

We could use:

```
foreign handler C_AToF(in pString as ZStringNative) returns CDouble \
    binds to "atof"
```

See the [Language Reference](#) for more information on C binding strings.

Callbacks

When used in the context of a foreign handler definition, a foreign handler type will cause automatic bridging of the LCB handler to a C function pointer which can be called directly by the native code.

The function pointers created in this fashion have lifetime equivalent to that of the calling context. In particular, for widgets they will last as long as the widget does, for all other module types they will last as long as the module is loaded.

For example, suppose you have the following function in `myLib`:

```
void qsort(void base, size_t nitems, size_t size, int (compar)(const void , const void))
```

```
void c_function_with_callback(void* p_context, bool(*func)(void* p_context))
```

This can be bound to using the following:

```
public foreign handler type CallbackType(in pContext as Pointer) returns CBool

foreign handler CFunctionWithCallback(in pContext as Pointer, in pHandler as
CallbackType) returns nothing \
    binds to "myLib>c_function_with_callback"

handler MyCallback(in pContext as Pointer) returns CBool
    ...

    return true
end handler

handler CallCFunctionWithCallback(in pContext as Pointer)
    CFunctionWithCallback(pContext, MyCallback)
end handler
```

Using compiled libraries

Extensions can include compiled libraries on which they depend. The libraries must be compiled for each platform and architecture they are required on and placed folders named with a platform ID in the extension code folder. The platform ID folder names are in the form:

```
<architecture>-<platform>[-<options>]
```

See the [platform ID](#) specification for more details.

Dynamic Linking

On all platforms with the exception of iOS devices only dynamically linked libraries are supported. Static linking is not yet supported in iOS simulator builds. On iOS 8+ devices dynamically linked frameworks (`.framework`) are supported.

Static Code Libraries for iOS Device Builds

Static libraries and frameworks for iOS must be compiled into a lcxext object. Use the [build-module-lcxext-ios.sh shell script](#) as an example of how to do so.

The script has the following inputs in order:

- The path to the module compiled to C++ with using `lc-compile` with the `--forcebuiltins --outputauxc OUTPUTFILE` options. The output file includes both the module and shims for any C foreign bindings. Use a `.cpp` extension so the compiler treats the file correctly.
- The deps file which is a text file listing required dependencies when linking the object where each line is in the form `{library | [weak-]framework} <name>`
- Lcxext output file path (must use the lcxext extension).
- The full name of the module. For example, `com.livecode.library.timezone`.
- The output file path for the ios module. This should be named `module.lcm` and be next to the lcxext object in the code folder. This is a dummy module we include in standalones that is just the name to be loaded as the actual module code is already linked into the executable.
- one or more full paths to static libraries. If linking a static framework the static library is located at `<name>.framework/<name>`

Java and Android Libraries

Libraries to load on android (`.jar` & `.aar`) may be placed in the code folder `jvm-android`.

Java libraries to load on all platforms supporting Java may be placed in the code folder `jvm`.

Example Directory Structure

```
module.lcm
manifest.xml
api.lcdoc
library.lcb
resources/image.png
code/universal-mac/library.dylib
code/x86-win/library.dll
code/x86-linux/library.so
code/x86_64-linux/library.so
code/armv6-android/library.so
code/jvm-android/library.aar
code/jvm/library.jar
code/universal-ios-iphones11.4/library.lcxext
code/universal-ios-iphones11.4/module.lcm
code/universal-ios-iphonesimulator11.4/library.dylib
```

Java

Java bindings are currently supported on Android, Mac and Linux. On desktop platforms the IDE will attempt to find the correct setting for the `JAVA_HOME` environment variable. As this can vary from distro to distro on Linux, this may fail - in this case ensure you set `JAVA_HOME` in script or using the message box.

Java Platform API

On any of the supported platforms, it is possible to bind to methods and fields of classes in the Java Platform API.

For example, take the [java.util.UUID class](#) . Here is a simple library which uses this class to return a new pseudo-randomly generated UUID as a string:

```
// The following comment block forms the top-level documentation of
// this library
/**
This library allows the creation of a UUID using the Java Platform
API.
*/
library com.livecode.library.javauuid

metadata title is "Java UUID"
metadata author is "LiveCode"
metadata version is "1.0.0"

// Bind to the static randomUUID() method of the java.util.UUID class
__safe foreign handler JNI_RandomUUID() returns JObject \
  binds to "java:java.util.UUID>randomUUID()Ljava/util/UUID;!static"

// Bind to the toString() instance method of the java.util.UUID class
__safe foreign handler JNI_UUIDToString(in pUUID as JObject) returns JString \
  binds to "java:java.util.UUID>toString()Ljava/lang/String;"

// Library public handler - this will be accessible from LiveCode
// when this library is loaded. The following comment block will
// be used to generate the documentation
/**
Returns a new random UUID.

Returns (String):
The string representation of a type 4 (pseudo randomly generated)
UUID.
*/
public handler GetRandomUUIDJava() returns String
  // Call the static randomUUID method to return an instance of
  // the UUID class
  variable tUUID as JObject
  put JNI_RandomUUID() into tUUID

  // Call the toString method on the UUID instance to obtain the
  // (java) string representation of the UUID
  variable tUUIDString as JString
  put JNI_UUIDToString(tUUID) into tUUIDString

  // Convert to a LiveCode String and return
  return StringFromJString(tUUIDString)
end handler

end library
```

Other binding examples

- Binding to a class constructor with no parameters:

```
__safe foreign handler CreateJavaObject() returns JObject binds to "java:java.lang.Object>new()"
```

- Binding to a class constructor with parameters:

`__safe foreign handler CreateJavaString(in pBytes as JByteArray) returns JString binds to "java:java.lang.String>new([B]"`

- Binding to a class instance method

`__safe foreign handler JavaStringsIsEmpty(in pString as JString) returns JBoolean binds to "java:java.lang.String>isEmpty()Z"`

- Binding to a class static method

`__safe foreign handler CallJavaAdd(in pLeft as JInt, in pRight as JInt) returns JInt binds to "java:java.lang.Math>addExact(JJ)!static"`

- Binding to a class field

safe foreign handler JavaCalendarSetTime(in pCalendar as JObject, in pTime as JLong) returns nothing binds to "java:java.util.Calendar>set.time(J)" safe foreign handler JavaCalendarGetTime(in pCalendar as JObject) returns JLong binds to "java:java.util.Calendar>get.time()J"

- Binding to a class constant

`__safe foreign handler GetJavaPi() returns JDouble binds to "java:java.lang.Math>get.PI()D!static"`

Android API

On Android it is possible to access the standard platform APIs using Java FFI.

<https://github.com/livecode/livecode/blob/develop/extensions/libraries/toast/toast.lcb>

Third-party libraries

The standalone builder will look for .jar files in the `code/jvm` folder of an extension package and load them automatically on startup of a standalone. Due to the way things currently work, testing extensions with jar files in the IDE is not supported. However it can be done by setting the CLASSPATH environment variable before the Java Virtual Machine is initialized. See [Custom Java classes](#) for an example of how to do this.

For android, the possibilities for including third-party libraries expands substantially. The `code/jvm-android` folder of an extension package can contain .jar and .aar files, and obviously the .jar files can make use of the Android API.

Classes in .jar files (either included directly, or within .aar packages) can be accessed using Java FFI in exactly the same way as classes in the Android API.

.aar packages can also contain resources and AndroidManifest.xml files - these are merged with the manifest and resources generated by LiveCode when building an Android application.

Custom Java classes

As .jar file code resources are supported, you can write and compile custom Java classes for use with an LCB extension.

Assuming the current folder is the root of an extension, consider the following java code, in a file `java/com/livecode/library/helloworldjava/HelloWorld.java`:


```
package com.livecode.library.helloworldjava;

public class HelloWorld
{
    public static String sayHello()
    {
        return "Hello World!"
    }
}
```

This can be compiled and packaged using java command-line tools:

```
javac java/com/livecode/library/helloworldjava/HelloWorld.java
jar cvf code/jvm/HelloWorld.jar -C java .
```

Now there should be a HelloWorld.jar file in the appropriate location relative to the extension.

In helloworld.lcb,

```
library com.livecode.library.helloworldjava

use com.livecode.foreign
use com.livecode.java

__safe foreign handler JNI_SayHello() returns JString \
    binds to
"java:com.livecode.library.helloworldjava.HelloWorld>sayHello()Ljava/lang/String;"

public handler SayHello() returns String
    return StringFromJString(JNI_SayHello())
end handler

end library
```

A standalone including this extension would be able to execute the script

```
answer SayHello()
```

to pop up an answer dialog with the text "Hello World!". Obviously that is a rather circuitous route when you can already do

```
answer "Hello World!"
```

but it does open up a lot of possibilities for interoperation with Java libraries.

One such application would be the ability to record .wav files on Android. The MediaPlayer class does not

come equipped with this functionality.

This can be compiled and packaged using java command-line tools, ensuring the appropriate android.jar is included in the dependencies (here we assume the location of the android toolchain is standard and the android-23 platform is downloaded):

```
javac -cp ~/android/toolchain/android-sdk/platforms/android-23/android.jar
java/src/com/livecode/library/androidwavrecorder/ExtAudioRecorder.java

jar cvf code/jvm-android/AndroidWavRecorder.jar -C java/src .
```

Now there should be a AndroiWavRecorder.jar file in the appropriate location relative to the extension.

Objective-C

The types used in Objective-C foreign handler declarations are the usual C types, plus a few special ones for handling Obj-C `id` objects. These are:

- `ObjcId` - an id with no implicit action on its reference count
- `ObjcRetainedId` - an id which is expected to already have been retained. (i.e. the caller or callee expects to receive it with +1 ref count)
- `ObjcAutoreleasedId` - an id which has been placed in the innermost autorelease pool before being returned to the caller

The `ObjcObject` type is a wrapper round the raw `ObjcId` types, which manages the lifetime of the obj-c object it contains. It should generally be used when using obj-c objects within LCB.

Objective-C binding strings mirror as closely as possible the structure of method declarations in Objective-C. For example, the class `NSSpeechSynthesizer` has methods

```
- (id)initWithVoice:(NSString *)voice;
- (BOOL)startSpeakingString:(NSString *)string;
```

In order to create an instance of the class, we first use the superclass `NSObject`'s

```
+ (id)alloc;
```

method to allocate an instance, and then call `initWithVoice:` on it:

```
foreign handler Objc_NSSpeechSynthesizerAlloc() returns ObjcRetainedId \
  binds to "objc:NSSpeechSynthesizer.+alloc"
```

The `+` indicates this is a class method, i.e. we don't require an instance of the class to call the method.

```
foreign handler Objc_NSSpeechSynthesizerInitWithVoice(in pSynthesizer as
ObjcRetainedId, in pVoice as optional ObjcId) returns optional ObjcRetainedId \
  binds to "objc:NSSpeechSynthesizer.-initWithVoice:"
```

The `-` here indicates this is an instance method, i.e. we require an instance of the class to call the method. This is always the first parameter. We can then create and initialize an `NSSpeechSynthesizer` using the following. We pass `nothing` to `initWithVoice:` to use the default voice.

```
handler CreateSpeechSynthesizer() returns ObjcObject
  variable tSynthesizer as ObjcObject
  put Objc_NSSpeechSynthesizerAlloc() into tSynthesizer

  return Objc_NSSpeechSynthesizerInitWithVoice(tSynthesizer, \
    nothing)
end handler
```

To speak, we would bind to the `startSpeakingString:` method:

```
foreign handler Objc_NSSpeechSynthesizerStartSpeaking(in pSynthesizer as ObjcId,
in pString as ObjcId) returns CBool \
  binds to "objc:NSSpeechSynthesizer.-startSpeakingString:"
```

Then expose a public library handler to enable calls from LiveCode.

```
private variable mSynthesizer as optional ObjcObject
public handler SynthesizeSpeech(in pString as String) returns Boolean
  if mSynthesizer is nothing then
    put CreateSpeechSynthesizer() into mSynthesizer
  end if

  Objc_NSSpeechSynthesizerStartSpeaking(mSynthesizer, \
    StringToNSString(pString))
end handler
```

Callbacks

Callbacks are usually handled by creating delegates. A class delegate handles the events associated with that class and marshalls the callback message.

Objective-C delegate objects with LCB implementations of protocol methods can be created using the `CreateObjcDelegate` and `CreateObjcInformalDelegate` handlers, provided by the `com.livecode.objc` module.

In order to create a delegate to handle a particular protocol method, pass in the protocol name as the first argument and the mapping from method names to LCB handlers as the second argument. For example, to create a `selectionChanged` message for an `NSTextView`, we need to create a handler

```
private handler DidChangeSelection(in pNotification as ObjcObject) returns nothing
    post "selectionChanged"
end handler
```

and create a `NSTextViewDelegate`:

```
variable tDelegate as optional ObjcObject
put CreateObjcDelegate( \
    "NSTextViewDelegate", \
    {"textViewDidChangeSelection": DidChangeSelection}, \
    ) into tDelegate
if tDelegate is not nothing then
    put tDelegate into mTextViewDelegate
end if
```

Optionally, a context parameter can be passed in at delegate creation time:

```
put CreateObjcDelegateWithContext( \
    "NSTextViewDelegate", \
    {"textViewDidChangeSelection": DidChangeSelectionContext}, \
    tContext) into tDelegate

if tDelegate is not nothing then
    put tDelegate into mTextViewDelegate
end if
```

In this case the context variable will be passed as first argument of the corresponding LCB callback:

```
private handler DidChangeSelectionContext(in pContext, in pNotification as
ObjcObject) returns nothing
    post "selectionChanged" with [pContext]
end handler
```

Some protocols consist of purely optional methods. In this case the information about the protocol's methods are not available from the objective-c runtime API. For this eventuality there are also handlers `CreateObjcInformalDelegate` and `CreateObjcInformalDelegateWithContext`.

These handlers take a list of foreign handlers as their first argument instead of a protocol name. The foreign handlers' information is used to resolve incoming selectors so that the desired LCB callback is called. For example the `NSSoundDelegate` protocol has only one method, and it is optional,

```
- (void)sound:(NSSound *)sound didFinishPlaying:(BOOL)aBool;
```

So in order to create an `NSSoundDelegate`, we need to create a list of foreign handlers, in this case just the following:

```
foreign handler NSSoundDidFinishPlaying(in pSound as ObjcId, in pDidFinish as
CChar) binds to "objc:.-sound:didFinishPlaying:"
```

and create the informal delegate

```
handler DidSoundFinish(in pSound as ObjcId, in pDidFinish as Boolean) returns
nothing
    if pDidFinish then
        post "soundFinished"
    end if
end handler

foreign handler Objc_SetSoundDelegate(in pSound as ObjcId, in pDelegate as ObjcId)
returns nothing \
    binds to "objc:NSSound.-setDelegate:"
...

variable tDelegate as optional ObjcObject
put CreateObjcInformalDelegate( \
    [NSSoundDidFinishPlaying], \
    {"textViewDidChangeSelection": DidChangeSelection}) \
    into tDelegate
end if
if tDelegate is not nothing then
    put tDelegate into mSoundDelegate
    Objc_SetSoundDelegate(tSound, tDelegate)
end if
```

Note: Delegate properties are usually 'assigned' rather than 'retained', so it is necessary to store them in module variables until they are no longer needed. Generally the pattern required is as follows:

```
handler OnOpen()
    -- Create native view and set native layer
    -- Set native view delegate property
    -- Store view and delegate in module vars
end handler

handler OnClose()
    -- Set native view delegate property to nothing
    -- Put nothing into view and delegate module vars
    -- Set native layer to nothing
end handler
```

For our speech synthesizer example, suppose we want to post a message when the speaking is finished.

In NSSpeechSynthesizer.h, we can see the definition of the NSSpeechSynthesizerDelegate:

```
@protocol NSSpeechSynthesizerDelegate <NSObject>
@optional
- (void)speechSynthesizer:(NSSpeechSynthesizer *)sender didFinishSpeaking:
(BBOOL)finishedSpeaking;
...
```

The first thing to note is that the protocol methods are all optional. This means that we have to use an informal delegate. To attach an LCB handler to the didFinishSpeaking event, we need to define both the foreign handler that binds to the protocol method and the LCB handler that should be called. Again this is an instance method of the delegate class, so the first parameter will be the delegate itself.

Again since the protocol method is optional, it will not be found if we bind to NSSpeechSynthesizerDelegate.-speechSynthesizer:didFinishSpeaking:, so it is necessary to use dynamic binding for the foreign handler (i.e. simply omitting the class in the binding string):

```
foreign handler Objc_SpeechSynthesizerDidFinishSpeaking(in pSynthesizerDelegate as
ObjcId, in pSynthesizer as ObjcId, in pFinished as CBool) returns nothing \
    binds to "objc:.-speechSynthesizer:didFinishSpeaking:"

handler OnDidFinishSpeaking(in pSynthesizerDelegate as ObjcId, in pSynthesizer as
ObjcId, in pFinished as CBool) returns nothing
    post "finishedSpeaking"
end handler

foreign handler Objc_SpeechSynthesizerSetDelegate(in pSynthesizer as ObjcId, in
pDelegate as ObjcId) returns nothing \
    binds to "objc:NSSpeechSynthesizer.-setDelegate:"

handler SetSpeechSynthesizerDelegate()
    variable tDelegate as ObjcObject
    put CreateObjcInformalDelegate( \
        [Objc_SpeechSynthesizerDidFinishSpeaking], \
        { "speechSynthesizer:didFinishSpeaking": \
            OnDidFinishSpeaking }) into tDelegate

    Objc_SpeechSynthesizerSetDelegate(mSynthesizer, tDelegate)
end handler
```

Using native views for widgets

The key piece of syntax for creating widgets that hook into native view objects is my native layer:

```
set my native layer to tView
```

The tView object is different on each platform, as is the method of specifying LCB handlers for native view event callbacks. We will use the example of a native button on each platform to illustrate this, where

possible.

The native layer should be created and destroyed every time the card the widget is on is opened and closed respectively. Therefore the pattern that should be adopted when writing a native widget is to use the `OnOpen` and `OnClose` messages as follows:

```
private variable mNativeLayer as optional Pointer
handler OnOpen()
    put CreateNativeLayer() into mNativeLayer
    set my native layer to mNativeLayer
end handler

handler OnClose()
    set my native layer to nothing
    put nothing into mNativeLayer
end handler
```

When using a native layer, a widget's `OnPaint` handler is not called. However it is recommended to provide some sort of placeholder `OnPaint` method to represent the widget when the native layer is not supported on the current platform.

Android

Native view

Native views on Android are classes derived from `android.view.View`. The Android button widget's native layer is an instance of the `android.widget.Button` class:

```
↳ android.view.View
  ↳ android.widget.TextView
    ↳ android.widget.Button
```

The following snippet shows how to bind to the button constructor and set the native layer:

```

// Bind to Android engine methods in order to fetch the application
// Context
__safe foreign handler _JNI_GetAndroidEngine() returns JObject \
    binds to
"java:com.runrev.android.Engine>getEngine()Lcom/runrev/android/Engine;!static"
__safe foreign handler _JNI_GetEngineContext(in pEngine as JObject) returns
JObject \
    binds to "java:android.view.View>getContext()Landroid/content/Context;"

// Bind to the android.widget.Button constructor
__safe foreign handler _JNI_ButtonNew(in pContext as JObject) returns JObject \
    binds to "java:android.widget.Button>new(Landroid/content/Context;)?ui"

// Store a reference to the native view
private variable mNativeView as optional JObject

handler CreateNativeLayer() returns Pointer
    // Fetch the application Context
    variable tContext as JObject
    put _JNI_GetEngineContext(_JNI_GetAndroidEngine()) into tContext

    // Create an instance of the android.widget.Button class
    put _JNI_ButtonNew(tContext) into mNativeView

    return PointerFromJObject(mNativeView)
end handler

handler SetNativeLayer()
    set my native layer to CreateNativeLayer()
end handler

```

View objects on Android always require the application Context in their constructors.

View objects should always be created on the UI thread.

Event handlers

A button widget is obviously not complete without an action callback. We want a message to be received by the widget object in LiveCode whenever the native view is clicked.

On Android, the general procedure is to attach a `Listener` to an object. In Java, `Listeners` are classes which implement a specified set of callback methods defined in that listener's interface. The general method of defining [interface callbacks on Android](#) is used to create listeners with LCB handler callbacks.

The following snippet shows how to attach an `OnClickListener` to the Button instance:


```

// Define handler type for button click callback. This matches
// the android.view.View.OnClickListener's
//      abstract void onClick(View v)
// method.
handler type ClickCallback(in pView as JObject) returns nothing

// Bind to the interface proxy method. Since there is only one
// method to be implemented for the OnClickListener interface,
// we just pass in a handler of the appropriate type.
__safe foreign handler _JNI_OnClickListener(in pHandler as ClickCallback) returns
JObject \
    binds to "java:android.view.View$OnClickListener>interface()"

// Bind to the method used to set the OnClickListener on the
// Button object.
__safe foreign handler _JNI_SetOnClickListener(in pButton as JObject, in pListener
as JObject) returns nothing \
    binds to
"java:android.view.View>setOnClickListener(Landroid/view/View$OnClickListener;)V?
ui"

// Actual handler (of type ClickCallback) that will be called
// when the button is clicked
handler OnButtonClick(in pView as JObject) returns nothing
    // The widget object in LiveCode will receive the posted message
    post "mouseUp"

    // Ensure the engine thread is notified of a pending event
    MCEngineRunLoopBreakWait()
end handler

// Store the OnClickListener object in a private variable
private variable mOnClickListener as optional JObject

// Set the OnClickListener
handler SetOnClickListener(in pButton as JObject) returns nothing
    // Pass the OnButtonClick handler in the call to create a
    // listener
    put _JNI_OnClickListener(OnButtonClick) into mOnClickListener

    // Set the OnClickListener on a Button instance
    _JNI_SetOnClickListener(pButton, mOnClickListener)
end handler

```

Sometimes listener interfaces have multiple callback methods, for different events. In this case, you must pass an array mapping callback method names to handlers so that the correct handler is called for each event. This can also be done with single-method interfaces, and indeed is the recommended style to use as it will continue to work even if additional callback methods are added to the interface in subsequent API levels.

Switching to this style would require two small tweaks to the above code:

```
// Bind to the interface proxy method.
__safe foreign handler _JNI_OnClickListener(in pMapping as Array) returns JObject
\
    binds to "java:android.view.View$OnClickListener>interface()"

...

// Set the OnClickListener
handler SetOnClickListener(in pButtonView as JObject) returns nothing
    // Map the OnButtonClick handler to the onClick method
    put _JNI_OnClickListener({"onClick":OnButtonClick}) \
        into mOnClickListener

    // Set the OnClickListener on a Button instance
    _JNI_SetOnClickListener(pButtonView, mOnClickListener)
end handler
```

Hooking up properties

Properties of Android native views can be set using methods of the view (or one of its parent classes). For example, to hook up the enabled property of the button, we bind to the `setEnabled` method of the parent `TextView` class:

```
__safe foreign handler _JNI_SetTextViewEnabled(in pView as JObject, in pValue as
JBoolean) returns nothing \
    binds to "java:android.view.View>setEnabled(Z)V?ui"

handler SetEnabled(in pButtonView as Pointer)
    _JNI_SetTextViewEnabled(pButtonView, my enabled)
end handler
```

iOS

Native view

Native views on iOS are classes derived from `UIView`. The iOS button widget's native layer is an instance of the `UIButton` class.

The following snippet shows how to bind to the button constructor and set the native layer:

```

// Define an alias for CULong
private type NSUInteger is CULong

// Bind to the UIButton class method buttonWithType:
private foreign handler ObjC_UIButtonButtonWithType(in pType as NSUInteger)
returns ObjcId binds to "objc:UIButton.+buttonWithType:?ui"

// For a standard push button we need the type to be
// UIButtonTypeSystem, which is 1
constant UIButtonTypeSystem is 1

// Store a reference to the native view
private variable mNativeView as optional ObjcObject
unsafe handler CreateNativeLayer() returns Pointer
    // Create an instance of the UIButton class
    variable tButtonInstance as ObjcObject
    put ObjC_UIButtonButtonWithType(UIButtonTypeSystem) \
        into mNativeView

    // Return the pointer
    return PointerFromObjcObject(mNativeView)
end handler

handler SetNativeLayer()
    set my native layer to CreateNativeLayer()
end handler

```

Event handlers

Views on iOS relay information about user interactions using either the [target-action paradigm](#) or [delegation](#)

In general, simple widgets will use the target-action paradigm. In general, refer to the documentation for the particular class you are wrapping. According to the [UIButton API](#), in the section 'Responding to Button Taps', the `UIButton` class uses the target-action paradigm.

So, we can post a message to the widget script object in response to the click action as follows:

```

// Bind to the UIButton addTarget:action:forControlEvents: method
private foreign handler ObjC_UIButtonAddTargetActionForControlEvents(in pObj as
ObjcId, in pTarget as ObjcId, in pAction as UIntPtr, in pControlEvents as
NSUInteger) returns nothing binds to "objc:UIButton.-
addTarget:action:forControlEvents:?ui"

// Actual handler that will be called when the button is clicked
handler OnButtonClick(in pSender as ObjcObject, in pContext as optional any)
returns nothing
    // The widget object in LiveCode will receive the posted message
    post "mouseUp"

    // Ensure the engine thread is notified of a pending event
    MCEngineRunLoopBreakWait()
end handler

// Store the action proxy object in a private variable
private variable mButtonProxy as optional ObjcObject

// The push button action type is UIControlEventTouchUpInside,
// which is 1 << 6. Constants cannot be evaluated expressions, so
// just define the bit shift
constant UIControlEventTouchUpInsideBit is 6

// Add the target action
unsafe handler AddTargetAction(in pButtonView as ObjcObject) returns nothing
    // Create a proxy for the button object. This will receive the
    // button events and call the provided callback handler
    put ObjcProxyGetTarget(OnButtonClick, nothing) into mButtonProxy

    // Add the target-action to the button view
    ObjC_UIButtonAddTargetActionForControlEvents(pButtonView, \
        mButtonProxy, ObjcProxyGetAction(), \
        1 shifted left by UIControlEventTouchUpInsideBit bitwise)
end handler

```

More complex views on iOS require creation of delegates. For example, there are several different user interactions possible with the `UITextField` view. In order to handle these interactions, we create a `UITextFieldDelegate` - see the [API](#) for the list of events that can be handled.

In this example we hook up `openField` and `textChanged` messages:

```

// Bind to the text field's delegate property setter
private foreign handler Objc_SetTextFieldDelegate(in pTextField as ObjcId, in
pDelegate as ObjcId) returns nothing \
    binds to "objc:UITextField.-setDelegate:?ui"

// Handler to be called when the field is opened
handler OnOpenField(in pField as ObjcId) returns nothing
    // The widget object in LiveCode will receive the posted message
    post "openField"

    // Ensure the engine thread is notified of a pending event
    MCEngineRunLoopBreakWait()
end handler

// Define the NSRange type, one of the parameters in the
// textFieldShouldChangeCharactersInRange method
public foreign type NSRange binds to "MCAggregateTypeInfo:ff"

// Handler to be called when the field text is changed
handler OnTextChanged(in pField as ObjcId, in pRange as NSRange, in pReplacement
as ObjcId) returns CBool
    // The widget object in LiveCode will receive the posted message
    post "textChanged"

    // Ensure the engine thread is notified of a pending event
    MCEngineRunLoopBreakWait()

    // Allow the text to be changed
    return true
end handler

// Store the delegate in a private variable
private variable mTextFieldDelegate as optional ObjcObject

// Set the delegate
unsafe handler SetTextFieldDelegate(in pTextFieldView as ObjcObject) returns
nothing
    // Create a delegate mapping the textFieldDidBeginEditing:
    // method to OnOpenField, and the
    // textFieldShouldChangeCharactersInRange: method to
    // OnTextChanged

    // First define the mapping
    variable tMapping as Array
    put { "textFieldDidBeginEditing:": OnOpenField, \
        "textField:shouldChangeCharactersInRange:replacementString:": \
        OnTextChanged } into tMapping

    put CreateObjcDelegate("UITextFieldDelegate", tMapping) \
        into mTextFieldDelegate

    Objc_TextFieldSetDelegate(pTextFieldView, mTextFieldDelegate)
end handler

```

Hooking up properties

Properties of iOS native views can be set using methods of the view (or one of its parent classes). For example, to hook up the enabled property:

```
foreign handler ObjC_UIButtonSetEnabled(in pObj as ObjcId, in pEnabled as CBool)
returns nothing \
  binds to "objc:UIButton.-setEnabled:?ui"

handler SetEnabled(in pButtonView as Pointer)
  ObjC_UIButtonSetEnabled(pButtonView, my enabled)
end handler
```

Mac

Native views on Mac are classes derived from `NSView`. The Mac button widget's native layer is an instance of the `NSButton` class.

The following snippet shows how to bind to the button constructor and set the native layer:

```

// Define an alias for CULong
private type NSUInteger is CULong

// Bind to the NSButton class method to allocate a new NSButton
private foreign handler ObjC_NSButtonAlloc() returns ObjcRetainedId binds to
"objc:NSButton.+alloc"

// Bind to the NSButton instance method to initialize an NSButton
private foreign handler ObjC_NSButtonInit(in pObj as ObjcId) returns ObjcId binds
to "objc:NSButton.-init"

// Bind to NSButton instance methods setButtonType:, setBezelStyle:,
// and -setBordered:
private foreign handler ObjC_NSButtonSetButtonType(in pObj as ObjcId, in pStyle as
NSUInteger) returns nothing binds to "objc:NSButton.-setButtonType:"
private foreign handler ObjC_NSButtonSetBezelStyle(in pObj as ObjcId, in pStyle as
NSUInteger) returns nothing binds to "objc:NSButton.-setBezelStyle:"
private foreign handler ObjC_NSButtonSetBordered(in pObj as ObjcId, in pBordered
as CBool) returns nothing binds to "objc:NSButton.-setBordered:"

// For a standard push button we need:
// buttonType to be NSMomentaryPushInButton = 7
// bezelStyle to be NSRoundedBezelStyle = 1
constant kNSMomentaryPushInButton is 7
constant kNSRoundedBezelStyle is 1

// Store a reference to the native view
private variable mNativeView as optional ObjcObject

unsafe handler CreateNativeLayer() returns Pointer
    // Create an instance of the NSButton class
    variable tButtonInstance as ObjcObject
    put ObjC_NSButtonInit(ObjC_NSButtonAlloc()) into mNativeView

    // Set the properties we need for a push button
    ObjC_NSButtonSetButtonType(mNativeView, kNSMomentaryPushInButton)
    ObjC_NSButtonSetBezelStyle(mNativeView, kNSRoundedBezelStyle)
    ObjC_NSButtonSetBordered(mNativeView, true)

    // Return the pointer
    return PointerFromObjcObject(mNativeView)
end handler

handler SetNativeLayer()
    set my native layer to CreateNativeLayer()
end handler

```

Views on Mac relay information about user interactions using either the [target-action paradigm](#) or [delegation](#)

In general, simple widgets will use the target-action paradigm. In general, refer to the documentation for the particular class you are wrapping. The `NSButton` uses the target-action paradigm.

So, we can post a message to the widget script object in response to the click action as follows:

```
private foreign handler ObjC_NSButtonSetTarget(in pObj as ObjcId, in pTarget as
ObjcId) returns nothing binds to "objc:NSButton.-setTarget:"
private foreign handler ObjC_NSButtonSetAction(in pObj as ObjcId, in pAction as
UIntPtr) returns nothing binds to "objc:NSButton.-setAction:"

// Actual handler that will be called when the button is clicked
handler OnButtonClick(in pSender as ObjcObject, in pContext as optional any)
returns nothing
    // The widget object in LiveCode will receive the posted message
    post "mouseUp"

    // Ensure the engine thread is notified of a pending event
    MCEngineRunLoopBreakWait()
end handler

// Store the action proxy object in a private variable
private variable mButtonProxy as optional ObjcObject

// Add the target action
unsafe handler AddTargetAction(in pButtonView as ObjcObject) returns nothing
    put ObjcProxyGetTarget(OnButtonClick, nothing) \
        into mButtonProxy
    ObjC_NSButtonSetTarget(pButtonView, mButtonProxy)
    ObjC_NSButtonSetAction(pButtonView, ObjcProxyGetAction())
end handler
```

More complex views on Mac require creation of delegates. See the discussion of delegates on iOS for more information.

Hooking up properties

Properties of Mac native views can be set using methods of the view (or one of its parent classes). For example, to hook up the enabled property:

```
foreign handler ObjC_NSButtonSetEnabled(in pObj as ObjcId, in pEnabled as CBool)
returns nothing \
    binds to "objc:NSButton.-setEnabled:"

handler SetEnabled(in pButtonView as Pointer)
    ObjC_NSButtonSetEnabled(pButtonView, my enabled)
end handler
```

Windows

Important: It is not yet possible to write a fully functional native widget on Windows, as there is no method to capture events on the native view and pass them back to LiveCode Builder. This will be addressed in future releases.

Native view

Native views on Windows are HWNDs. The `CreateWindowEx` function is used to create HWNDs with specified properties. See the [API documentation](#) for details on the parameters taken by `CreateWindowEx`.

The following snippet shows how to bind to `CreateWindowEx`, and use it to set the native layer to a button HWND:

```
// Alias the types used in native calls
public type DWORD is CULong
public type WIDESTRING is ZStringUTF16
public type HINSTANCE is Pointer
public type HMENU is Pointer
public type HWND is Pointer

// Bind to the CreateWindowEx function. It is defined in the
// user32.dll library (hence `user32>` in the binding string).
// The CreateWindowEx function definition is preceded by the WINAPI
// macro, which means it uses the stdcall calling convention (hence
// `!stdcall` in the binding string)
foreign handler CreateWindow(in pEx as DWORD, \
    in pString as WIDESTRING, in pLabel as WIDESTRING, \
    in pStyle as DWORD, in pX as CInt, in pY as CInt, \
    in pWidth as CInt, in pHeight as CInt, in pParent as HWND, \
    in pMenu as optional HMENU, in pInstance as optional HINSTANCE, \
    in pParam as optional Pointer) \
    returns HWND binds to "user32>CreateWindowExW!stdcall"

private variable mNativeView as optional Pointer

// Constants for the CreateWindow style parameter. We construct the
// DWORD by taking the bitwise or of the styles that are
// appropriate.

// BS_DEFPUSHBUTTON is the push button style constant
constant BS_DEFPUSHBUTTON is 1

// WS_CHILD is the child window style constant. If this is not used
// then `CreateWindowEx` creates a new main window.
constant WS_CHILD is 1073807361

// CreateWindowEx requires the parent view pointer to be passed in
unsafe handler CreateNativeView(in pParentView as Pointer)
    // Compute the style DWORD
    variable tStyle as DWORD
    put WS_CHILD bitwise or BS_DEFPUSHBUTTON into tStyle

    // Pass in the predefined BUTTON window class
    variable tWindowClass as String
    put "BUTTON" into tWindowClass

    // Call CreateWindow, with empty/default values for all other
    // parameters
    put CreateWindow(0, tWindowClass, "", tStyle, 0, 0, 0, 0, \
```

```

        pParentView, nothing, nothing, nothing) \
        into mNativeView

    return mNativeView
end handler

unsafe handler SetNativeLayer()
    // Get the parent view pointer
    variable tParentView as Pointer
    MCWidgetGetMyStackNativeView(tParentView)

    set my native layer to CreateNativeLayer(tParentView)
end handler

```

Event handlers

As the general callback method has not yet been implemented on Windows, unfortunately widgets that require user interaction are not yet possible. However, native widgets can be written that have only properties and no user interaction, such as a progress indicator or label field.

Hooking up properties

Properties of Windows native views can be set using functions in the user32 dll. For example, to hook up the enabled property of a Windows button:

```

foreign handler EnableWindow(in pView as Pointer, in pEnable as CInt) \
    returns CInt binds to "user32>EnableWindow!stdcall"

handler SetEnabled(in pButtonView as Pointer)
    EnableWindow(pButtonView, my enabled)
end handler

```

Linux

Important: It is not yet possible to write a fully functional native widget on Linux, as there are some issues with event handling and focus. This will be addressed in future releases.

Native view

Native views on Linux are `GtkPlug`s. A `GtkSocket` is used internally to render the view from the process in which it is running. In most instances you will want to embed a `GtkWidget` in a `GtkPlug`.

The following snippet shows how to bind to `gtk_button_new_with_label`, and use it to set the native layer to a suitable gtk plug id:

```

// Bind to various useful functions in libgtk. We need to create a
// GTK plug, and GTK button, add the button to the plug and then
// show them.
foreign handler GTK_PlugNew(in pType as CUInt) returns Pointer \
    binds to "c:libgtk-x11-2.0.so>gtk_plug_new"

foreign handler GTK_ButtonNewWithLabel(in pLabel as ZStringNative) returns Pointer
binds to \
    "c:libgtk-x11-2.0.so>gtk_button_new_with_label"

foreign handler GTK_ContainerAdd(in pContainer as Pointer, in pWidget as Pointer)
returns nothing \
    binds to "c:libgtk-x11-2.0.so>gtk_container_add"

foreign handler GTK_WidgetShow(in pWidget as Pointer) returns nothing binds to \
    "c:libgtk-x11-2.0.so>gtk_widget_show"

// The actual native layer will be set to the plug id
foreign handler GTK_PlugGetId(in pPlug as Pointer) returns Pointer \
    binds to "c:libgtk-x11-2.0.so>gtk_plug_get_id"

// Store references to both the plug and button
private variable mPlug as optional Pointer
private variable mNativeView as optional Pointer

unsafe handler CreateNativeView() returns Pointer
    // Create a new default plug
    put GTK_PlugNew(0) into mPlug

    // Create a button with empty label
    put GTK_ButtonNewWithLabel("") into mNativeView

    // Add the button to the plug
    GTK_ContainerAdd(mPlug, mNativeView)

    // Ensure both button and plug are visible
    GTK_WidgetShow(mNativeView)
    GTK_WidgetShow(mPlug)

    // Return the plug window id
    return GTK_PlugGetId(mPlug)
end handler

unsafe handler SetNativeLayer()
    set my native layer to CreateNativeLayer()
end handler

```

Event handlers

Event callbacks are pretty simple on Linux, as you can pass a (foreign) handler into `g_signal_connect_data` directly:

```
// Define the callback foreign handler type
public foreign handler type ClickCallback(in pWidget as Pointer, in pContext as
optional Pointer) returns nothing

// Bind to g_signal_connect_data to connect a foreign handler to
// a gtk signal
foreign handler GTK_SignalConnect(in pObj as Pointer, in pEvent as ZStringNative, \
    in pHandler as ClickCallback, in pData as optional Pointer, \
    in pNotify as optional Pointer, in pFlags as CUInt) returns CULong \
    binds to "c:libgtk-x11-2.0.so>g_signal_connect_data"

// Actual handler that will be called when the button is clicked
handler OnButtonClick(in pWidget as Pointer, in pContext as optional Pointer)
returns nothing
    // The widget object in LiveCode will receive the posted message
    post "mouseUp"
end handler

unsafe handler SignalConnect(in pButtonView as Pointer) returns nothing
// Connect the foreign handler to the clicked signal. All other
// parameters are empty/default
GTK_SignalConnect(pButtonView, "clicked", OnButtonClick, \
    nothing, nothing, 0)
end handler
```

Hooking up properties

Properties of Linux native views can be set using the GTK Widget API. For example, to hook up the enabled property of a Linux button (called *sensitive* in the GTK API):

```
foreign handler GTK_WidgetSetSensitive(in pWidget as Pointer, in pValue as CBool)
returns nothing \
    binds to "c:libgtk-x11-2.0.so>gtk_widget_set_sensitive"

handler SetEnabled(in pButtonView as Pointer)
    GTK_WidgetSetSensitive(pButtonView, my enabled)
end handler
```

HTML5

Native view

Native views on HTML5 are document elements, which can be created by evaluating JavaScript and returning the element.

The following snippet shows how to evaluate JavaScript using the `EvalJavaScript` handler from the `emscripten` module to create a new button element, and set the native layer of the widget.

```
// Store the button element pointer
private variable mNativeView as optional Pointer

handler CreateNativeView() returns Pointer
    // Create an HTML5 button using JavaScript
    put EvalJavaScript("document.createElement('button')") \
        into mNativeView

    return PointerFromJSObject(mNativeView)
end handler

handler SetNativeLayer() returns nothing
    set my native layer to CreateNativeView()
end handler
```

Event handlers

Event handlers can be attached to native views again by evaluating JavaScript:

```
// Utility function for adding an event listener
handler AddJSEventHandler(in pElement as JSObject, in pEvent as String, in
pHandler as JSObject)
    EvalJavaScriptWithArguments("arguments[0].addEventListener(arguments[1],
arguments[2]);", [pElement, pEvent, pHandler])
end handler

// Actual handler that will be called when the button is clicked
handler OnJSClick(pEvent as JSObject) returns nothing
    // The widget object in LiveCode will receive the posted message
    post "mouseUp"
end handler

// Store a reference to the JS handler
private variable mOnClickHandler as optional Pointer

handler AttachEventHandler(in pButtonView as Pointer)
    // Get a JS function that wraps the LCB callback
    put HandlerAsJSFunction(OnJSClick) into mOnClickHandler

    // Add the handler to the button for the click event
    AddJSEventHandler(pButtonView, "click", mOnClickHandler)
end handler
```

Hooking up properties

On HTML5, properties of the native view are manipulated by evaluating JavaScript. For example, the enabled of the HTML5 button can be implemented as follows:

```

handler SetEnabled(in pButtonView as JSObject)
  // Set button enabled using JavaScript
  EvalJavaScriptWithArguments( \
    "arguments[0].disabled=arguments[1];", \
    [pButtonView, not my enabled])
end handler

```

The extension package

The extension package is a zip file (with extension `.lce`) which, in addition to the actual compiled lcb file, contains various other components related to the extension: icons, documentation, guides, a manifest, sample stacks, extension resources and native code.

Packages can be uploaded to the extensions store to be shared with other users of LiveCode.

Documenting Your Extension

Extensions can provide an API (Dictionary) and User Guide as part of the installed package. They are installed and viewable through the LiveCode Dictionary stack.

API (Dictionary)

-
1. Open the documentation stack from the menubar
 2. API chooser - Choose which API to view. By default, LiveCode 8.0 comes with the LiveCode Script API and the new LiveCode Builder API. Any extension that contains an API is displayed here.
 3. Category Filter - Click on the tags to filter the API entries.
 4. Search Filter - Enter a text string to filter the API entries by a specific search string
 5. History - Move back and forward through your API navigation history
 6. Entry List - Select the entry to display from this list
 7. Entry - The entry describing the syntax

User Guide

-
1. Click on the "Guide" tab at the top of the documentation stack
 2. Select the guide you wish to view

Adding API Markup

Any extension can include an API. To do so, either add a file called *api.lcdoc* to your widget folder alongside the other widget files or markup your source code inline.

Marking up your scripts is simple and follows a similar model to other documentation formats.

Consider the following handler

```
public handler myHandler(in pString as String, in pNumber as Number)
  # Code
end handler
```

To add an entry to the API for this handler, place a formatted comment above the handler definition:

```
/**
Summary: Use this handler to do an action

Parameters:
pString: This parameter does x
pNumber: This parameter does y

Description:
# Markdown Title
Here is a full description in markdown for how this function works.
Once again, any GitHub flavoured markdown is accepted.
*/
public handler myHandler(in pString as String, in pNumber as Number)
  # Code
end handler
```

The LiveCode API parser will combine the comment items with other data pulled directly from your handler definition such as name, type and some parameter details.

For a full list of acceptable documentation elements please see the LiveCode Documentation Format guide.

Important: Documentation is only added to the LiveCode dictionary when an extension is installed in the IDE.

Adding a User Guide

Any extension can include a user guide. To do so, add a markdown file called *guide.md* to your widget folder alongside the other widget files.

All GitHub flavoured markdown is accepted.

Packaging Extensions

Packaging extensions is easy. Simply open the "Extension Builder" stack:

Tools > Extension Builder

Load your **.lcb** file and click "Build Package". A new file will appear in the same directory as your main file with the extension **.lce**. This is your final package file ready for upload to the extensions portal.

Package requirements

A valid package can be built from a minimum set of files:

```

widget.lcb // Widget source file
support/icon.png // 20x40 png image
support/icon@extra-high.png // 40x80 png image

```

The Extension Store

Once you have created and packaged your extension you can make it available to the LiveCode Community via the [Extension Store](#), in the [widgets](#) section.

Registering as an Extension Developer

The first step is to [register](#) as an Extension Developer.

- Log in to your LiveCode Account
- Select the Register option under the Developer section

LiveCode Extensions must have unique identifiers. These identifiers use reverse domain notation.

You can sign up on this page for a LiveCode Community Developer ID. Having a developer ID will allow you to use extension identifiers of the form: `community.livecode.developerID.extensionName`.

Choose the Developer ID you want to use here. If your chosen ID is already in use you will be asked to choose a different one.

Uploading Extensions to the Store

To upload a packaged extension to the store log in to your LiveCode account and go to the [Extensions](#) section of the Developer area.

Any extensions you have uploaded will be shown here, and you have the option to add new extensions.

- Click the "Add New Extension" button (if available)
- Drag your packaged extension (`.lce`) file into the upload area
- When the extension has uploaded it will appear in your list

Activating your Extension

When an extension is first uploaded it is not active so will not show in the Extension Store.

To activate your extension click "activate" under Status. At this stage you will be warned if any information is missing and you can add it by clicking the "Edit" button.

Once your extension is activated its status will change to "Active".

The Extension Store in the Extension Manager

You can view and download extensions via the Extension Store. To see the extensions that are available to download open the Extension Manager from the LiveCode Tools menu and select the "Store" tab.

You can also download extensions from the [Widget Store](#) on the LiveCode website then install the downloaded extension using the Extension Manager.

Other ways to extend the Built-in Capabilities

There are many other ways to extend LiveCode. This section explains how to run shell commands, start other applications, read and write to processes, execute AppleScript, VBScript, send and respond to AppleEvents and communicate between multiple LiveCode-based processes. It also tells you where to get information to create external commands and functions (code written in lower level languages).

Communicating with other process and applications

Reading and writing to the command shell

Use the **shell** function to run shell commands and return the result. The following example displays a directory listing on Mac OS X:

```
answer shell("ls")
```

And this example stores a directory listing in a variable on Windows:

```
put shell("dir") into tDirectory
```

On Windows systems you can prevent a terminal window from being displayed by setting the **hideConsoleWindows** global property to true.

You can choose a different shell program by setting the **shellPath** global property. By default this is set to `/bin/sh` on Mac OS X and Linux and `command.com` on Windows.

Tip: The shell function blocks LiveCode until it is completed. If you want to run a shell command in the background, write the shell script to a text file then execute it with the `launch command`.

Launching other applications

Use the **launch** command to launch other applications, documents or URLs. To launch an application, supply the full path to the application. The following example opens a text document with TextEdit on OS X:

```
launch "/Users/someuser/Desktop/text document.rtf" with  
"/Applications/TextEdit.app"
```

Tip: To get the path to an application, use the `answer file` command to select the application then copy it into your script. Run this in the message box:

```
`answer file "Select an application"; put it`
```

To open a document with the application it is associated with use the **launch document** command.

```
launch document "C:/My document.pdf"
```

To open a URL in the default web browser, use the **launch URL** command.

```
launch URL "<http://www.livecode.com/>"
```

For more information on launching URLs and details on how to render web pages within LiveCode, see the Transferring Information guide.

Closing another application

Use the **kill process** command to send a signal to another application, to close it or to force it to exit. For more details, see the LiveCode Dictionary.

Communicating with other processes

Use the **open process** command to open an application or process you want to read and write data from. You can then read from the process with the **read from process** command and write to it with the **write to process** command. To close a process you have opened, use the **close process** command. The **openProcesses** returns a list of processes you have opened and the **openProcessIDs** returns the process IDs of each one. For more details see the *LiveCode Dictionary*.

Using AppleScript and VBScript (Open Scripting Architecture or

Windows Scripting Host)

To execute commands using AppleScript on Mac OS or VBScript on Windows, use the **do as** command. **do as** also allows you to use any other *Open Scripting Architecture* languages on Mac OS or languages installed into the *Windows Scripting Host* on Windows. To retrieve a list of the available installed languages, use the **alternateLanguages**.

For example, to execute an AppleScript that brings the Finder on OS X to the front, enter the following into a field:

```
tell application "Finder" activate end tell
```

Then run:

```
do field 1 as "appleScript"
```

To retrieve a result from commands executed using `do as`, use the `the result` function. Any error message will also be returned in `the result`. The following example displays `the result` of an

addition performed using VBScript:

```
do "result = 1 + 1" as "vbscript"
answer the result
```

For more information on the do as command, see the LiveCode Dictionary.

AppleEvents

To send an AppleEvent, use the **send to program** command.

If LiveCode receives an AppleEvent it will send an **appleEvent** message to the current card. Intercept this message to perform actions such as handling a request to quit your application or opening a document. The following example shows how you could handle a request to quit:

```
on appleEvent pClass, pID, pSender
  if pClass & pID is "aevtquit" then
    -- call a function that prompts the user to save changes
    put checkSaveChanges() into tOkToQuit
    -- returns false if the user presses "cancel"
    if tOkToQuit is true then quit
    else exit appleEvent
  end if
end appleEvent
```

To retrieve additional information passed with the appleEvent use the **request appleEvent data** command. The following example shows how you could handle a request to open a stack:

```
on appleEvent pClass, pID, pSender
  --appleEvent sent when stack is opened from the finder
  if pClass & pID is " aevtodoc " then
    -- get the file path(s)
    request AppleEvent data
    put it into tFilesList
    repeat for each line l in tFilesList
      go stack l
    end repeat
  end if
end appleEvent
```

For more details see the *LiveCode Dictionary*.

Using Local sockets

If you want to communicate between local applications a common technique that can be used without code changes on all the platforms LiveCode supports, is to open a local socket and communicate using that. You should choose a port number that is not used by a standard protocol – typically a high number.

This technique is commonly used when you want to create multiple programs that run independently but communicate with each other. It is a viable technique for running background tasks and provides a

straightforward way to create an application that behaves as if threaded – i.e. with benefits of multiple threads. You can design your application such that additional instances can be launched to perform processing, data transfer or other intensive activities. Modern OSes will allocate each application to an appropriate processor core. By using socket messaging to communicate with each one you can keep your main application's user interface responsive and display status information. The following example shows you how to open a socket to the local machine:

```
open socket to "127.0.0.1:10000" with message gotConnection
```

A detailed discussion of how to create a protocol using sockets can be found in the Transferring Information guide.

Tip: To simplify communication between multiple LiveCode programs, consider writing a simple library that sends and receives a handler name together with parameter data. To call a handler in the other LiveCode program, send the handler name and data to the library. The library will send the data over a socket. In the receiving program intercept the incoming data from the socket and use it to call the appropriate message with the parameter data received.

Externals - code written in lower level languages

LiveCode provides an external interface which allows you to extend it using a lower level language (often C). For example, if you have preexisting code that performs processing in a lower level language, you can write a user interface in LiveCode and then call this library by writing a simple wrapper around it using LiveCode's externals interface. LiveCode supports transmitting data to and from externals, as well as drawing into image objects within LiveCode windows, manipulating the player object, and more.

Important: Much of what is provided by the externals API is now supported by the foreign function interface in LiveCode Builder, where it is not necessary to write any glue code in C. Whilst the externals interface is still supported, LCB is now the recommended way to wrap native code for LiveCode.

Note: Some aspects of the built in functionality are supplied in the form of externals. These include the SSL library, the database library, the revBrowser library, zip library and XML libraries. These libraries can be included in a standalone application, or excluded if they are not needed – saving disk space.

The Externals SDK

Occasionally LiveCode Builder may not be what you need to create your extension. We provide a legacy developer kit for writing externals in C++ which includes documentation and examples.

The following newsletter articles will help you get started:

- [External Writing for the Uninitiated – Part 1](#)
- [External Writing for the Uninitiated – Part 2](#)
- [Writing Externals for Linux with 2.9 or later](#)

LiveCode Builder Bytecode Reference

Introduction

The LiveCode Builder Virtual Machine is a infinite register machine with a high-level and verifiable bytecode.

All bytecode runs in the context of a module instance with executing handlers forming a stack of activation frames. Each frame contains an array of registers, the first part of which are parameters followed by handler-local variables and bytecode block registers.

Most bytecode operations operate directly on registers, access to module level definitions (handlers, constants, variables) are indirected through the fetch and store operations.

Each bytecode operation has an address which can be jumped to using the jump operations.

Operations

Jump

```
jump <label>
```

The jump operation sets the address of the next instruction to execute to that identified by .

Jump If False

```
jump_if_false <register>, <label>
```

The jump_if_true operation checks the value in and jumps to if it is 'false'.

If is a runtime error if does not contain a boolean.

Jump If True

```
jump_if_true <register>, <label>
```

The jump_if_true operation checks the value in and jumps to if it is 'true'.

If is a runtime error if does not contain a boolean.

Assign Constant

```
assign_constant <register>, <constant>
```

The assign_constant operation copies into . Here can be the nothing literal, the true or false literal, an integer literal, a real literal, a string literal, a list literal or array literal.

If is typed then it is a runtime error if the type of does not conform to that type.

Assign

```
assign <dst-register>, <src-register>
```

The assign operation copies the value in to .

If is typed then it is runtime error if the type of the value in does not conform to that type.

Return

```
return [ <result-reg> ]
```

The return operation exits the current handler, returning to the instruction after the invoke operation which created it. If present, the value in is copied as the return value. If not present then the return value will be 'nothing'. Additionally any values in out or inout parameters are copied back to the caller.

If is not present, then the return value will be 'nothing'.

It is a runtime error if the type of the return value does not conform to the return type of the current handler's signature.

It is a runtime error if any out or inout parameters are unassigned at the point of return.

Invoke

```
invoke <handler>, <result-reg>, <arg1-reg>, ..., <argn-reg>
```

The invoke operation creates a new frame copying values from the argument registers for any in or inout parameters. It then starts executing the bytecode attached to , a definition. The return value is placed into the register .

If is a runtime error if the number of arguments provided is different from the signature of

If it a runtime error if for in and inout parameters, the contents of does not conform to the type of the parameter required by the signature.

Invoke Indirect

```
invoke <handler-reg>, <result-reg>, <arg1-reg>, ..., <argn-reg>
```

The invoke indirect operation functions identically to the invoke operation except that it calls the handler in .

It is a runtime error if does not contain a handler value.

Fetch

```
fetch <dst-register>, <definition>
```

The fetch operation copies the value of `<src-reg>` into `<dst-reg>`. The `<dst-reg>` may be a variable, constant or handler. In the case of a handler, a handler value is created. It is a runtime error if the type of the value of `<src-reg>` does not conform to the type of `<dst-reg>`.

Store

```
store <src-register>, <definition>
```

The store operation copies the contents of `<src-reg>` into `<dst-reg>`. The `<dst-reg>` must be a module level variable. It is a runtime error if the type of the value in `<src-reg>` does not conform to the type of `<dst-reg>`.

Assign List

```
assign_list <dst-reg>, <element1-reg>, ..., <elementn-reg>
```

The `assign_list` operation builds a list value from up to `<elementn-reg>`. It is a runtime error if the type of `<dst-reg>` does not conform to list.

Assign Array

```
assign_array <dst-reg>, <key1-reg>, <value1-reg>, ..., <keyn-reg>, <valuen-reg>
```

The `assign_array` operation builds an array value from each key value pair `<key1-reg>, <value1-reg>`, up to `<keyn-reg>, <valuen-reg>`. It is a runtime error if the type of `<dst-reg>` does not conform to array.

Reset

```
reset <reg>
```

The reset operation performs default initialization of `<reg>`. If the type of `<reg>` has no default value, it reverts to unassigned.

LiveCode Builder Language Reference

Introduction

LiveCode Builder is a variant of the current LiveCode scripting language (LiveCode Script) which has been designed for 'systems' building. It is statically compiled with optional static typing and direct foreign code interconnect (allowing easy access to APIs written in other languages).

Unlike most languages, LiveCode Builder has been designed around the idea of extensible syntax. Indeed, the core language is very small - comprising declarations and control structures - with the majority of the language syntax and functionality being defined in modules.

Note: It is an eventual aim that control structures will also be extensible, however this is not the case in the current incarnation).

The syntax will be familiar to anyone familiar with LiveCode Script, however LiveCode Builder is a great deal more strict - the reason being it is intended that it will eventually be compilable to machine code with the performance and efficiency you'd expect from any 'traditional' programming language. Indeed, over time we hope to move the majority of implementation of the whole LiveCode system over to being written in LiveCode Builder.

Note: One of the principal differences is that type conversion is strict - there is no automatic conversion between different types such as between number and string. Such conversion must be explicitly specified using syntax (currently this is using things like ... *parsed as number* and ... *formatted as string*).

Tokens

The structure of tokens is similar to LiveCode Script, but again a little stricter. The regular expressions describing the tokens are as follows:

- **Identifier:** `[A-Za-z][A-Za-z0-9.]*`
- **Integer:** `[0-9]+`
- **Binary Integer:** `0b[01]+`
- **Hexadecimal Integer:** `0x[0-9a-fA-F]+`
- **Real:** `[0-9]+."[0-9]+([eE][-+]?[0-9]+)?`
- **String:** `"[^\n\r"]*"'`
- **Separator:** Any whitespace containing at least one newline

Strings use backslash ('\') as an escape - the following are understood:

- `\n`: LF (ASCII 10)
- `\r`: CR (ASCII 13)
- `\t`: TAB (ASCII 9)
- `\q`: quote ""
- `**\u{X...X}`: character with unicode codepoint U+X...X - any number of nibbles may be specified, but any values greater than 0x10FFFF will be replaced by U+FFFD.
- `\`: backslash '\'

Note: The presence of '.' in identifiers are used as a namespace scope delimiter.

Note: Source files are presumed to be in UTF-8 encoding.

Comments

LiveCode Builder supports single line comments, which begin with `//` or `--` and extend to the end of the line. There are also block comments, which begin with `/*` and end with `*/`, and can span multiple lines.

- **Single-line comment:** `(--|//)[^\n\r]*`
- **Block comment:** `/*(.*?)*/`

Note: A block comment that spans multiple lines terminates the line of code that it begins on.

Line continuation

A LiveCode builder statement or declaration can be continued onto multiple lines of code by placing the line continuation character `\` at the end each line.

- **Line continuation:** `[\t]*(\n|\r\n|\r)`

Note: Tab and space characters are allowed after the `\` and before the newline, but no other characters.

Note: A line continuation cannot occur within a comment.

Case-Sensitivity

At the moment, due to the nature of the parser being used, keywords are all case-sensitive and reserved. The result of this is that, using all lower-case identifiers for names of definitions should be avoided. However, identifiers *are* case-insensitive - so a variable with name `pFoo` can also be referenced as `PFOO`, `PfOO`, `pfoO` etc.

Aside: The current parser and syntax rules for LiveCode Builder are constructed at build-time of the LiveCode Builder compiler and uses *bison* (a standard parser generator tool) to build the parser. Unfortunately, this means that any keywords have to be reserved as the parser cannot distinguish the use of an identifier in context (whether it is a keyword at a particular point, or a name of a definition).

It is highly recommended that the following naming conventions be used for identifiers:

- **tVar** - for local variables
- **pVar** - for in parameters
- **rVar** - for out parameters
- **xVar** - for inout parameters
- **mVar** - for global variables in widgets
- **sVar** - for global variables in libraries
- **kConstant** - for constants
- Use identifiers starting with an uppercase letter for handler and type names.

By following this convention, there will not be any ambiguity between identifiers and keywords. (All keywords are all lower-case).

Note: When we have a better parsing technology we will be evaluating whether to make keywords case-insensitive as well. At the very least, at that point, we expect to be able to make all keywords unreserved.

Typing

LiveCode Builder is a typed language, although typing is completely optional in most places (the only exception being in foreign handler declarations). If a type annotation is not specified it is simply taken to be the most general type *optional any* (meaning any value, including nothing).

The range of core types is relatively small, comprising the following:

- **nothing**: the single value *nothing*
- **Boolean**: one of *true* or *false*
- **Integer**: any integral numeric value (size limitations apply)
- **Real**: any numeric value (size and accuracy limitations apply)
- **Number**: any integer or real value
- **String**: a sequence of UTF-16 code units
- **Data**: a sequence of bytes
- **List**: a sequence of any values
- **Array**: a mapping from strings to values
- **any**: a value of any type

Additionally, all types can be annotated with **optional**. An optional annotation means the value may be the original type or nothing.

Note: The current compiler does not do type-checking; all type-checking happens at runtime. However, this is being worked on so there will soon be a compiler which will give you type errors at compile-time.

Modules

```
Module
: 'module' <Name: Identifier> SEPARATOR
  { ( Definition | Metadata | Import ) SEPARATOR }
'end' 'module'
```

The smallest compilable unit of LiveCode Builder is the module. Each module is uniquely named using reverse DNS notation, and the names of modules are considered to live in a global namespace.

A module is a collection of public and private definitions, including constants, variables, types and handlers.

A module may depend on another module through import. An imported modules public definitions become accessible to the importing module.

Note: For integration with the existing LiveCode system, there are two module variants which may be used. Widgets (use 'widget' instead of 'module') and Libraries (use 'library' instead of 'module'). A widget appears in LiveCode as a control, whilst a library adds all its public handlers to the bottom of the message path.

Metadata

```
Metadata
: 'metadata' <Name: Identifier> 'is' <Value: String>
```

The metadata clauses allow a set of key-values to be encoded in the compiled module. These are not used in compilation or execution, but may be used by the system loading and using the module.

At the moment, the following keys are understood:

- title: a human-readable name for the module
- description: a simple description of the module's purpose
- version: a string in the form X.YZ (with X, Y and Z integers) describing the modules version
- author: the name of the author of the module
- os: the operating systems where the module can be used
- platforms: the platforms where the module can be used

Note: The current metadata mechanism is unlikely to remain part of the language. It is intended that it will be replaced by a package description file, which will allow modules to be grouped together with other resources.

Imports

```
Import
: 'use' <Name: Identifier>
```

The use clauses allow a module to refer to another module by importing all the target module's public definitions into its namespace.

The name of the module specified must be its full name, e.g. com.livecode.canvas.

A module may use any other module, as long as doing so does not cause a cycle in the dependency graph.

Note: The current IDE and extension installation system does not yet implement arbitrary dependencies - the only dependencies it understands are those which are builtin to the system (e.g. com.livecode.canvas). However, you can still write and test out modules with dependencies locally - they just cannot be uploaded to the extensions portal.

Definitions

```
Definition
: ( 'public' | 'private' ) ConstantDefinition
| ( 'public' | 'private' ) TypeDefinition
| ( 'public' | 'private' ) HandlerTypeDefinition
| ( 'public' | 'private' ) VariableDefinition
| ( 'public' | 'private' ) HandlerDefinition
| ( 'public' | 'private' ) ForeignHandlerDefinition
| PropertyDefinition
| EventDefinition
```

Definitions are what are used to define usable entities in the language. All definitions are named using a unique identifier (so you cannot have two definitions with the same name).

Definitions can be either *public* or *private* (the default is private

- so there is no need to explicitly specify that). Public definitions are available when the module is used by another module whereas private definitions can only be used within the module.

Note: Properties and events are, by their nature, always public as they define things which

only make sense to access from outside.

Note: When writing a library module, all public handlers are added to bottom of the message path in LiveCode Script.

Constants

```
ConstantDefinition
: 'constant' <Name: Identifier> is <Value: Expression>
```

A constant definition defines a named constant. The value can be any expression which depends on only on constant values to evaluate.

Types

```
TypeDefinition
: 'type' <Name: Identifier> 'is' <TypeOf: Type>
```

A type definition defines an alias, it names the given type with the given Name, allowing the name to be used instead of the type.

```
Type
: <Name: Identifier>
| 'optional' <Target: Type>
| 'any'
| 'nothing'
| 'Boolean'
| 'Integer'
| 'Real'
| 'Number'
| 'String'
| 'Data'
| 'Array'
| 'List'
| 'Pointer'
```

A type clause describes the kind of value which can be used in a variable or parameter.

If a type is an identifier, then this is taken to be a named type defined in a type definition clause.

An optional type means the value can be either the specified type or nothing. Variables which are of optional type are automatically initialized to nothing.

The remaining types are as follows:

- **any:** any value
- **Boolean:** a boolean value, either the value *true* or *false*.
- **Integer:** any integer number value
- **Real:** any real number value
- **Number:** any number value
- **String:** a sequence of UTF-16 code units
- **Data:** a sequence of bytes

- **Array:** a map from string to any value (i.e. an associative array, just like in LiveCode Script)
- **List:** a sequence of any value
- **nothing:** a single value *nothing* (this is used to describe handlers with no return value - i.e. void)
- **Pointer:** a low-level pointer (this is used with foreign code interconnect and shouldn't be generally used).

Note: *Integer* and *Real* are currently the same as *Number*.

Note: In a subsequent update you will be able to specify lists and arrays of fixed types. For example, *List of String*.

Note: In a subsequent update you will be able to define record types (named collections of values - like structs in C) and handler types (allowing dynamic handler calls through a variable - like function pointers in C).

Handler Types

```
HandlerTypeDefinition
: [ 'foreign' ] 'handler' 'type' <Name: Identifier> '(' [ ParameterList ] ')' [
'returns' <ReturnType: Type> ]
```

A handler type definition defines a type which can hold a handler. Variables of such types can hold handlers (just like function pointers in C) which allows them to be called dynamically.

If the handler type is defined as foreign then automatic bridging to a C function pointer will occur when the type appears as the type of a parameter in a foreign handler definition.

Note: Passing an LCB handler to a foreign function requires creation of a function pointer. The lifetime of the function pointer is the same as the widget or module which created it.

Record Types

```
RecordTypeDefinition
: 'record' 'type' <Name: Identifier> SEPARATOR
{ RecordTypeFieldDefinition }
'end' 'record'

RecordTypeFieldDefinition
: <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A record type definition defines a type that consists of 0 or more named fields, each with its own optional type.

Variables

```
VariableDefinition
: 'variable' <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A variable definition defines a module-scope variable. In a widget module, such variables are per-widget (i.e. instance variables). In a library module, there is only a single instance (i.e. a private global variable).

The type specification for the variable is optional, if it is not specified the type of the variable is *optional*

any meaning that it can hold any value, including being nothing.

Variables whose type has a default value are initialized to that value at the point of definition. The default values for the standard types are:

- **optional**: nothing
- **Boolean**: false
- **Integer**: 0
- **Real**: 0.0
- **Number**: 0
- **String**: the empty string
- **Data**: the empty data
- **Array**: the empty array
- **List**: the empty list
- **nothing**: nothing

Variables whose type do not have a default value will remain unassigned and it is a checked runtime error to fetch from such variables until they are assigned a value.

Handlers

```
HandlerDefinition
: [ 'unsafe' ] 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [
'returns' <ReturnType: Type> ] SEPARATOR
  { Statement }
'end' 'handler'
```

Handler definitions are used to define functions which can be called from LiveCode Builder code, invoked as a result of events triggering in a widget module, or called from LiveCode Script if public and inside a library module.

There is no distinction between handlers which return a value and ones which do not, apart from the return type. Handlers can be called either in expression context, or in statement context. If a handler which returns no value (it is specified as *returns nothing*) is called in expression context then its value is *nothing*.

```
ParameterList
: { Parameter , ',' }

Parameter
: ( 'in' | 'out' | 'inout' ) <Name: Identifier> [ 'as' <ParamType: Type>
```

The parameter list describes the parameters which can be passed to the handler. Handlers must be called with the correct number of parameters, using expressions which are appropriate to the mode.

An in parameter means that the value from the caller is copied to the parameter variable in the callee handler.

An out parameter means that no value is copied from the caller, and the value on exit of the callee handler is copied back to the caller on return.

Note: It is a checked runtime error to return from a handler without ensuring all non-optional 'out' parameters have been assigned a value. However, this will only occur for typed variables whose type does not have a default value as those which do will be default

initialized at the start of the handler.

An inout parameter means that the value from the caller is copied to the parameter variable in the callee handler on entry, and copied back out again on exit.

The type of parameter is optional, if no type is specified it is taken to be *optional any* meaning it can be of any type.

Note: Only assignable expressions can be passed as arguments to inout or out parameters. It is a checked compile-time error to pass a non-assignable expression to such a parameter.

If 'unsafe' is specified for the handler, then the handler itself is considered to be unsafe, and may only be called from other unsafe handlers or unsafe statement blocks.

Foreign Handlers

```
ForeignHandlerDefinition
: 'foreign' 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [ 'returns'
<ReturnType: Type> ) ] 'binds' 'to' <Binding: String>
```

A foreign handler definition binds an identifier to a handler defined in foreign code.

The last parameter in a foreign handler declaration may be '...' to indicate that the handler is variadic. This allows binding to C functions such as printf.

Note: No bridging of types will occur when passing a parameter in the non-fixed section of a variadic argument list. You must ensure the arguments you pass there are of the appropriate foreign type (e.g. CInt, CDouble).

There are a number of types defined in the foreign, java and objc modules which map to the appropriate foreign type when used in foreign handler signatures.

There are the standard machine types (defined in the foreign module):

- Bool maps to an 8-bit boolean
- Int8/SInt8 and UInt8 map to 8-bit integers
- Int16/SInt16 and UInt16 map to 16-bit integers
- Int32/SInt32 and UInt32 map to 32-bit integers
- Int64/SInt64 and UInt64 map to 64-bit integers
- IntSize/SIntSize and UIntSize map to the integer size needed to hold a memory size
- IntPtr/SIntPtr and UIntPtr map to the integer size needed to hold a pointer
- NaturalSInt and NaturalUInt map to 32-bit integers on 32-bit processors and 64-bit integers on 64-bit processors
- NaturalFloat maps to the 32-bit float type on 32-bit processors and the 64-bit float (double) type on 64-bit processors

There are the standard C primitive types (defined in the foreign module)

- CBool maps to 'bool'
- CChar, CSChar and CUChar map to 'char', 'signed char' and 'unsigned char'
- CShort/CSShort and CUShort map to 'signed short' and 'unsigned short'
- CInt/CSInt and CUInt map to 'signed int' and 'unsigned int'
- CLong/CLong and CULong map to 'signed long' and 'unsigned long'
- CLongLong/CSLongLong and CULongLong map to 'signed long long' and 'unsigned long long'
- CFloat maps to 'float'
- CDouble maps to 'double'

There are types specific to Obj-C types (defined in the objc module):

- ObjcObject wraps an obj-c 'id', i.e. a pointer to an objective-c object
- ObjcId maps to 'id'
- ObjcRetainedId maps to 'id', and should be used where a foreign handler argument expects a +1 reference count, or where a foreign handler returns an id with a +1 reference count.

Note: When an ObjcId is converted to ObjcObject, the id is retained; when an ObjcObject converted to an ObjcId, the id is not retained. Conversely, when an ObjcRetainedId is converted to an ObjcObject, the object takes the +1 reference count (so does not retain); when an ObjcObject is put into an ObjcRetainedId, a +1 reference count is taken (so does retain).

There are aliases for the Java primitive types (defined in the java module)

- JBoolean maps to Bool
- JByte maps to Int8
- JShort maps to Int16
- JInt maps to Int32
- JLong maps to Int64
- JFloat maps to Float32
- JDouble maps to Float64

All the primitive types above will implicitly bridge between corresponding high level types:

- CBool and Bool bridge to and from Boolean
- All integer and real types bridge to and from Number

Other LCB types pass as follows into foreign handlers:

- any type passes an MCValueRef
- nothing type passes as the null pointer
- Boolean type passes an MCBooleanRef
- Integer type passes an MCNumberRef
- Real type passes an MCNumberRef
- Number type passes an MCNumberRef
- String type passes an MCStringRef
- Data type passes an MCDataRef
- Array type passes an MCArrayRef
- List type passes an MCProperListRef

Finally, the Pointer type passes as void * to foreign handlers. If you want a pointer which can be null, then use optional Pointer - LCB will throw an error if there is an attempt to map from the null pointer value to a slot with a non-optional Pointer type.

Modes map as follows:

- in mode is just pass by value
- out mode passes a pointer to a variable of one of the above types, the variable is uninitialized on entry
- inout mode passes a pointer to a variable of one of the above types, the variable is initialized to a value on entry

If the return type is of a Ref type, then it must be a copy.

If an out parameter is of a Ref type, then it must be a copy (on exit)

If an inout parameter is of a Ref type, then its existing value must be released, and replaced by a copy (on exit).

The binding string for foreign handlers is language-specific and currently supported forms are explained in the following sections.

Foreign handlers' bound symbols are resolved on first use and an error is thrown if the symbol cannot be found.

Foreign handlers are always considered unsafe, and thus may only be called from unsafe context - i.e. from within an unsafe handler, or unsafe statement block.

The C binding string

The C binding string has the following form:

```
"c:[library>][class.]function[!calling][?thread]"
```

Here *library* specifies the name of the library to bind to (if no library is specified a symbol from the engine executable is assumed).

Here *class* is currently unused.

Here *function* specifies the name of the function symbol to bind to (on Windows, the symbol must be unadorned, and so exported from the library by listing it in a DEF module).

Here *calling* specifies the calling convention which can be one of:

- default
- stdcall
- thiscall
- fastcall
- cdecl
- pascal
- register

All but `default` are Win32-only, and on Win32 `default` maps to `cdecl`. If a Win32-only calling convention is specified on a non-Windows platform then it is taken to be `default`.

Here *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

The Obj-C binding string

The Obj-C binding string has the following form:

```
"objc:[library>][class.](+|-)method[?thread]"
```

Here *library* specifies the name of the library or framework to bind to (if no library is specified a symbol from the engine executable or a library it is linked to is assumed).

Here *class* specifies the name of the class containing the method to bind to. If the method is an instance method, the class can be omitted, creating a 'dynamic binding', i.e. just resolving the selector.

Here *method* specifies the method name to bind to in standard Obj-C selector form, e.g.

`addTarget:action:forControlEvents:`. If the method is a class method then prefix it with '+', if it is an instance method then prefix it with '-'.

Here *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

The Java binding string

The Java binding string has the following form:

```
"java:[className>][functionType.]function[!calling][?thread]"
```

Here *className* is the qualified name of the Java class to bind to.

Here *functionType* is either empty, or `get` or `set`, which are currently used for getting and setting member fields of a Java class.

For example

```
"java:java.util.Calendar>set.time(J)"
"java:java.util.Calendar>get.time()J"
```

are binding strings for setting and getting the `time` field of a `Calendar` object.

Here *function* specifies the name of the method or field to bind to. The function `new` may be used to call a class constructor. *function* also includes the specification of function signature, according to the [standard rules for forming these](#) when calling the JNI.

The function `interface` may be used on Android to create an interface proxy - that is an instance of a generic Proxy class for a given interface. This effectively allows LCB handlers to be registered as the targets for java interface callbacks, such as event listeners.

The foreign handler binding to such a function takes a value that should either be a `Handler` or an `Array` - if it is a `Handler`, the specified listener should only have one available callback. If the listener has multiple callbacks, an array can be used to assign handlers to each. Each key in the array must match the name of a callback in the listener. The specified handlers must match the callback's parameters and return type, using `JSONObject` where primitive type parameters are used.

Overloaded methods in the interface are not currently supported.

For example:

```

handler type ClickCallback(in pView as JObject) returns nothing

foreign handler _JNI_OnClickListener(in pHandler as ClickCallback) returns JObject
binds to "java:android.view.View$OnClickListener>interface()"

foreign handler _JNI_SetOnClickListener(in pButton as JObject, in pListener as
JObject) returns nothing binds to
"java:android.view.View>setOnClickListener(Landroid/view/View$OnClickListener;)V"

public handler ButtonClicked(in pView as JObject) returns nothing
    post "buttonClicked"
    MCEngineRunLoopBreakWait()
end handler

public handler SetOnClickListenerCallback(in pButton as JObject)
    unsafe
        variable tListener as JObject
        put _JNI_OnClickListener(ButtonClicked) into tListener
        _JNI_SetOnClickListener(pButton, tListener)
    end unsafe
end handler

```

or

```

handler type MouseEventCallback(in pMouseEvent as JObject) returns nothing

foreign handler _JNI_MouseListener(in pCallbacks as Array) returns JObject binds
to "java:java.awt.event.MouseListener>interface()"

foreign handler _JNI_SetMouseListener(in pJButton as JObject, in pListener as
JObject) returns nothing binds to
"java:java.awt.Component>addMouseListener(Ljava/awt/event/MouseListener;)V"

public handler MouseEntered(in pEvent as JObject) returns nothing
    post "mouseEnter"
    MCEngineRunLoopBreakWait()
end handler

public handler MouseExited(in pEvent as JObject)
    -- do something on mouse enter
end handler

public handler SetMouseListenerCallbacks(in pJButton as JObject)
    variable tArray as Array
    put MouseEntered into tArray["mouseEntered"]
    put MouseExited into tArray["mouseExited"]
    unsafe
        variable tListener as JObject
        put _JNI_MouseListener(tArray) into tListener
        _JNI_SetMouseListener(pJButton, tListener)
    end unsafe
end handler

```

Important: On Android, interface callbacks are *always* run on the engine thread. This means JNI local references from other threads (in particular the UI thread) are unavailable. Therefore it is not advised to do anything using the JNI in interface callbacks.

Here *calling* specifies the calling convention which can be one of:

- instance
- static
- nonvirtual

Instance and nonvirtual calling conventions require instances of the given Java class, so the foreign handler declaration will always require a Java object parameter.

Here, *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

Warning: At the moment it is not advised to use callbacks that may be executed on arbitrary threads, as this is likely to cause your application to crash.

Properties

```
PropertyDefinition
: 'property' <Name: Identifier> 'get' <Getter: Identifier> [ 'set' <Setter:
Identifier> ]
```

Property definitions can only appear in widget modules. They define a property which can be accessed from LiveCode Script in the usual way (e.g. *the myProperty of widget 1*).

Both getter and setter clauses can use either a variable or handler identifier. If a variable identifier is used, then the property value is fetched (and stored) from that variable. If a handler identifier is used then a handler is called instead.

A getter handler must take no arguments and return a value. A setter handler must take a single argument and return no value.

The set clause is optional. If it is not present then the property is read-only.

Events

```
EventDefinition
: 'event' <Name: Identifier> '(' [ ParameterList ] ')' [ 'as' <ReturnType: Type>
]
```

Event definitions define a callable handler which calls back to the environment.

Note: Whilst events can be defined they currently cannot be used. To send a message to the LiveCode Script environment use the *dispatch* command which allows sending messages to arbitrary LiveCode Script objects.

Statements

```
Statement
: VariableStatement
| IfStatement
| RepeatStatement
| ThrowStatement
| ReturnStatement
| PutStatement
| SetStatement
| GetStatement
| CallStatement
| BytecodeStatement
| UnsafeStatement
```

There are a number of built-in statements which define control flow, variables, and basic variable transfer. The remaining syntax for statement is defined in auxiliary modules.

Variable Statements

VariableStatement

```
: 'variable' <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A variable statement defines a handler-scope variable. Such variables can be used after the variable statement and up to the end of the current statement block, but not before.

Variables whose type have a default value are initialized with that value at the point of definition in the handler. See the main Variables section for the defaults of the standard types.

Note: It is a checked runtime error to attempt to use a variable whose type has no default before it is assigned a value.

The type specification for the variable is optional, if it is not specified the type of the variable is *optional* any meaning that it can hold any value, including being nothing.

If Statements**IfStatement**

```
: 'if' <Condition: Expression> 'then' SEPARATOR
  { Statement }
[ { 'else' 'if' <Condition: Expression> 'then' SEPARATOR
  { Statement } }
[ 'else' SEPARATOR
  { Statement } ]
'end' 'if'
```

The if statement enables conditional execution based on the result of an expression which evaluates to a boolean.

Each block of code in an if/else if/else statement defines a unique scope for handler-local variable definitions.

Note: It is a checked runtime error to use an expression which does not evaluate to a boolean in any condition expression.

Repeat Statements**RepeatStatement**

```
: RepeatHeader SEPARATOR
  { Statement }
'end' 'repeat'
| 'next' 'repeat'
| 'exit' 'repeat'
```

RepeatHeader

```
: 'repeat' 'forever'
| 'repeat' <Count: Expression> 'times'
| 'repeat' 'while' <Condition: Expression>
| 'repeat' 'until' <Condition: Expression>
| 'repeat' 'with' <Counter: Identifier> 'from' <Start: Expression> ( 'up' |
'down' ) 'to' <Finish: Expression> [ 'by' <Step: Expression> ]
| 'repeat' 'for' 'each' <Iterator> 'in' <Container: Expression>
```

The repeat statements allow iterative execute of a sequence of statements.

The block of code present in a repeat statement defines a unique scope for handler-local variable definitions.

The **repeat forever** form repeats the body continually. To exit the loop, either an error must be thrown, or **exit repeat** must be executed.

The **repeat times** form repeats the body Count number times. If Count evaluates to a negative integer, it is taken to be zero.

Note: It is a checked runtime error to use an expression not evaluating to a number as the Count.

The **repeat while** form repeats the body until the Condition expression evaluates to false.

Note: It is a checked runtime error to use an expression not evaluating to a boolean as the Condition.

The **repeat until** form repeats the body until the Condition expression evaluates to true.

Note: It is a checked runtime error to use an expression not evaluating to a boolean as the Condition.

The **repeat with** form repeats the body until the Counter variable reaches or crosses (depending on iteration direction) the value of the Finish expression. The counter variable is adjusted by the value of the Step expression on each iteration. The start, finish and step expressions are evaluated before the loop starts and are not re-evaluated. The Counter variable must be declared before the repeat statement.

Note: It is a checked runtime error to use expressions not evaluating to a number as Start, Finish or Step.

The **repeat for each** form evaluates the Container expression, and then iterates through it in a custom manner depending on the Iterator syntax. For example:

```
repeat for each char tChar in "abcdef"
  -- do something with tChar
end repeat
```

The **next repeat** statement terminates the current iteration of the enclosing loop and starts the next iteration of the loop, or exits if currently on the last iteration.

The **exit repeat** statement terminates the current iteration of the enclosing loop, continuing execution at the statement after the enclosing loop's **end repeat**.

Throw Statements

```
ThrowStatement
: 'throw' <Error: Expression>
```

The **throw** statement causes an error to be raised. This causes execution to terminate, and the error is passed back to environment.

The Error expression must be an expression that evaluates to a string.

Note: There is currently no try / catch mechanism in LiveCode Builder, so throwing an error will cause the error to be raised in LiveCode Script in the appropriate context.

Return Statements

```
ReturnStatement
: 'return' [ <Value: Expression> ]
```

The **return** statement causes execution of the current handler to end, and control return to the caller.

If a Value expression is specified, it is evaluated and returned as the result of the handler call.

Note: It is a checked runtime error for a value returned from a handler to not match the return type of the handler it is in.

Transfer Statements

```
PutStatement
: 'put' <Value: Expression> into <Target: Expression>

SetStatement
: 'set' <Target: Expression> 'to' <Value: Expression>
```

The **put** and **set** statements evaluate the Value expression and assign the resulting value to the Target expression.

The target expression must be assignable.

Note: It is a checked runtime error for the source value's type to not be compatible with the target expression's type.

```
GetStatement
: 'get' <Value: Expression>
```

The **get** statement evaluates the Value expression and returns it as the result of the statement. The value is subsequently available by using **the result** expression.

Call Statements

```
CallStatement
: <Handler: Identifier> '(' [ <Arguments: ExpressionList> ] ')'
```

The call statement executes a handler with the given arguments.

The Handler identifier must be bound to either a handler or foreign handler definition.

The Arguments are evaluated from left to right and passed as parameters to the variable.

Note: It is a checked runtime error for the types of 'in' and 'inout' arguments to not match the declared types of the handler's parameters.

Any parameters of 'out' type are not evaluated on entry, but assigned to on exit.

Any parameters of 'inout' type are evaluated on entry, and assigned on exit.

Note: It is a checked compile-time error to pass non-assignable expressions to parameters which are of either 'out' or 'inout' type.

The return value of a handler is subsequently available by using **the result** expression.

Note: All handlers return a value, even if it is nothing. This means that calling a handler will always change **the result**.

Bytecode Statements

```
BytecodeStatement
: 'bytecode' SEPARATOR
  { BytecodeOperation }
  'end'

BytecodeOperation
: <Label: Identifier> ':'
| 'register' <Name: Identifier> [ 'as' <Type: Type> ]
| <Opcode: Identifier> { BytecodeArgument , ',' }

BytecodeArgument
: ConstantValueExpression
| Identifier
```

The bytecode statement allows bytecode to be written directly for the LiveCode Builder Virtual Machine.

Bytecode operation arguments can either be a constant expression, the name of a definition in current scope, the name of a register, or the name of a label in the current bytecode block. The exact opcodes and allowed arguments are defined in the LiveCode Builder Bytecode Reference.

Labels are local to the current bytecode block, and can be used as the target of one of the jump instructions.

Register definitions define a named register which is local to the current bytecode block. Registers are the same as handler-local variables except that they do not undergo default initialization.

Bytecode statements are considered to be unsafe and can only appear inside unsafe handlers or unsafe statement blocks.

Note: Bytecode blocks are not intended for general use and the actual available operations are subject to change.

Unsafe Statements

```
UnsafeStatement
: 'unsafe' SEPARATOR
  { Statement }
  'end' 'unsafe'
```

The unsafe statement allows a block of unsafe code to be written in a safe context.

In particular, calls to unsafe handlers (including all foreign handlers) and bytecode blocks are allowed in unsafe statement blocks but nowhere else.

Expressions

```
Expression
: ConstantValueExpression
| VariableExpression
| ResultExpression
| ListExpression
| ArrayExpression
| CallExpression
```

There are a number of expressions which are built-in and allow constant values, access to call results, list construction and calls. The remaining syntax for expressions is defined in auxiliary modules.

Constant Value Expressions

```
ConstantValueExpression
: 'nothing'
| 'true'
| 'false'
| INTEGER
| REAL
| STRING
```

Constant value expressions evaluate to the specified constant value.

The **nothing** expression evaluates to the nothing value and can be assigned to any optional typed variable.

The **true** and **false** expressions evaluate to boolean values.

The INTEGER and REAL expressions evaluate to numeric values.

The STRING expression evaluates to a string value.

Constant value expressions are not assignable.

Variable Expressions

```
VariableExpression
: <Name: Identifier>
```

Variable expressions evaluate to the value of the specified variable.

Variable expressions are assignable.

Result Expressions

```
ResultExpression
: 'the' 'result'
```

The result expression evaluates to the return value of the previous (executed) non-control structure statement.

Result expressions are not assignable.

List Expressions

```
ListExpression
: '[' [ <Elements: ExpressionList> ] '']'
```

A list expression evaluates all the elements in the expression list from left to right and constructs a list value with them as elements. Each expression is converted to `optional any` when constructing the list.

The elements list is optional, so the empty list can be specified as `[]`.

List expressions are not assignable.

Array Expressions

```
ArrayExpression
: '{' [ <Contents: ArrayDatumList> ] '}'
ArrayDatumList
: <Head: ArrayDatum> [ ',' <Tail: ArrayDatumList> ]
ArrayDatum
: <Key: Expression> ':' <Value: Expression>
```

An array expression evaluates all of the key and value expressions from left to right, and constructs an **Array** value as appropriate. Each key expression must evaluate to a **String**. Each value expression is converted to `optional any` when constructing the array.

The contents are optional, so the empty array can be written as `{}`.

Array expressions are not assignable.

Call Expressions

```
CallExpression
: <Handler: Identifier> '(' [ <Arguments: ExpressionList> ] ')'
```

A call expression executes a handler.

Its use is identical to a call statement, except that the return value of the handler is the value of the expression, rather than being available as **the result**.

Note: Handlers which return no value (i.e. have nothing as their result type) can still be used in call expressions. In this case the value of the call is **nothing**.

Namespaces

Identifiers declared in a module are placed in a scope named using the module name. This allows disambiguation between an identifier declared in a module and an identical one declared in any of its imports, by using a fully qualified name.

For example:

```

module com.livecode.module.importee

public constant MyName is "Importee"
public handler GetMyName() returns String
    return MyName
end handler

public type MyType is Number

end module

module com.livecode.module.usesimport

use com.livecode.module.importee

public constant MyName is "Uses Import"
public handler GetMyName() returns String
    return MyName
end handler
public type MyType is String

handler TestImports()
    variable tVar as String
    put MyName into tVar
    -- tVar contains "Uses Import"

    put com.livecode.module.importee.MyName into tVar
    -- tVar contains "Importee"

    put com.livecode.module.usesimport.MyName into tVar
    -- tVar contains "Uses Import"

    put com.livecode.module.importee.GetMyName() into tVar
    -- tVar contains "Importee"

    variable tVarMyType as MyType
    put tVar into tVar1 -- valid

    variable tVarImportedType as com.livecode.module.importee.MyType
    put tVar into tVarImportedType -- compile error
end handler

end module

```

Experimental Features

Warning: This section describes current language features and syntax that are considered experimental and unstable. They are likely to change or go away without warning.

Safe Foreign Handlers

```
SafeForeignHandler
: '__safe' 'foreign' 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [
'returns' <ReturnType: Type> ) ] 'binds' 'to' <Binding: String>
```

By default foreign handlers are considered unsafe and thus can only be used in unsafe blocks, or unsafe handlers. However, at the moment it is possible for a foreign handler to actually be safe if it has been explicitly written to wrap a foreign function so it can be easily used from LCB.

Specifically, it is reasonable to consider a foreign handler safe if it conforms to the following rules:

- Parameter types and return type are either ValueRefs, or bridgeable types
- Return values of ValueRef type are retained
- If the function fails then MCErrorsThrow has been used
- 'out' mode parameters are only changed if the function succeeds
- A return value is only provided if the function succeeds

Examples of foreign handlers which can be considered safe are all the foreign handlers which bind to syntax in the LCB standard library.

Foreign Aggregate Types

C-style aggregates (e.g. structs) can now be accessed from LCB via the new aggregate parameterized type. This allows calling foreign functions which has arguments taking aggregates by value, or has an aggregate return value.

Aggregate types are foreign types and can be used in C and Obj-C foreign handler definitions. They bridge to and from the List type, allowing an aggregate's contents to be viewed as a sequence of discrete values.

Aggregate types are defined using a `foreign type` clause and binding string. e.g.

```
public foreign type NSRect binds to "MCAggregateTypeInfo:qqqq"
```

The structure of the aggregate is defined by using a sequence of type codes after the ':', each type code represents a specific foreign (C) type:

Char	Type
a	CBool
b	CChar
c	CUChar
C	CSShort
d	CUShort
D	CSShort
e	CUInt
E	CSInt

Char	Type
f	CULong
F	CSLong
g	CULongLong
G	CSLongLong
h	UInt8
H	SInt8
i	UInt16
I	SInt16
j	UInt32
J	SInt32
k	UInt64
K	SInt64
l	UIntPtr
L	SIntPtr
m	UIntSize
M	SIntSize
n	Float
N	Double
o	LCUInt
O	LCSInt
p	NaturalUInt
P	NaturalSInt
q	NaturalFloat
r	Pointer

When importing an aggregate to a List, each field in the aggregate is also bridged, except for Pointer types which are left as Pointer. When exporting an aggregate from a List, each element is bridged to the target field type.

Note: Any foreign type binding to an aggregate must be public otherwise the type will not work correctly.

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](#) *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` domain 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

Prefix	Context	Meaning
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](#) 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(3.1415, 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(pProperties["showBorder"])
  end if
  if "opaque" is among the keys of pProperties then
    setShowBackground(pProperties["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_control_property

-- ...

metadata borderColor.editor is "com.livecode.pi.color
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

LiveCode Documentation Format Reference

Introduction

LiveCode uses a custom documentation format for the dictionary and for extension APIs. The format has very few special characters and is intended to maximise readability and modifiability.

Format

Documentation for a LiveCode API or documentation entry consists of a series of elements and their values, in the following form:

elementName : *content*

There are several standard types of *content*; which one may be used depends on the *elementName*.

Element names

Elements are optional unless otherwise specified.

Name (required)

The name of the API entry. The name must be a single word.

Note: For inline docs, the name element is automatically generated

Synonyms

A comma-delimited list of synonyms for this API entry.

Type (required)

The type of this API entry. One of the following: (API-level)

- command
- function
- property
- message
- constant
- keyword
- control structure
- operator
- statement
- expression

(Glossary-level)

- library
- widget
- glossary
- object

The glossary-level entries have part of their content generated from the API-level entries.

Note: For inline docs, the type element is automatically generated

Syntax (required for API-level entries)

A description of how to use or call this facet of the API. May just be the name itself, for keywords etc. An entry may have a number of Syntax elements, if there are variations within the same entry.

The following can be used to specify livecode syntax:

- [optional]
- [repeated optional ...]
- { variant 1 | variant 2 }
- <parameterName>

For example, the syntax for the first variant of the `split` command:

```
split <variableToSplit> {by | using | with} <primaryDelimiter> [and
<secondaryDelimiter>]
```

describes the following possible usages:

- `split tVar by comma`
- `split tVar using return`
- `split tVar with "a"`
- `split tVar by comma and return`
- `split tVar using the rowdel and the columndel`
- `split tVar with tDelim1 and tDelim2`

The syntax for `answer file with type`:

```
answer file[s] <prompt> [with <defaultPath>] [with type <types> [or type <types>
...]] [titled <windowTitle>] [as sheet]
```

gives an example of the repetition notation, namely it allows something like

```
answer files with type tType1 or type tType2 or type tType3 or type tType4
```

Note: For inline docs, the Syntax elements are automatically generated

All parameters described in the parameters section should appear in angle brackets in the syntax.

Summary (required)

A summary of the API entry being documented. The summary should be a single line. Anything more in-depth should be included in the description.

Associated

A comma-delimited list of names of other API or docs entries. Each one must be of type **object**,

glossary, **library**, or **widget**. It is used to generate the docs for the entries to which it is associated.

In particular, every API entry in a library should have that library name in its Associated list.

Introduced

The LiveCode version in which the API entry was first available.

OS

A comma-delimited list specifying which operating systems the API entry is available for. One or more of the following:

- mac
- windows
- linux
- iOS
- android
- RPi
- html5

Platforms

A comma-delimited list specifying which platforms the API entry is available for. One or more of the following:

- desktop
- server
- web
- mobile

Example

Example elements should show how API entries are used in context. Each example will be displayed in the documentation viewer as a multiline code block. Where possible, an example should be entirely self contained, so that it is runnable as a block of code without any additions.

An API entry can have any number of Example elements.

Parameters

The parameters element itself does not do anything, but can be helpful for readability as a precursor to the parameter elements themselves. Parameter elements are specified in the following way:

paramName (paramType) : paramDescription

The param type is optional, although may be helpful to include, and is required for further formatting of parameter descriptions to take place. The three types that will instigate further formatting are:

- enum
- set
- array

A parameter of **enum** type is one that expects one of a specified list of values. Similarly, a parameter of **set** type is one that expects a comma-delimited list of one or more of a specified list of values. After the param description of the **enum** or **set** type, these values should be listed in the following way


```
- value1 : description of value1
- value2 : description of value2
- ...
```

etc.

After the description parameter of **array** type, the format of the array may optionally be specified in the following way:

```
{
  key1 (keyType) : description of key1
  value1 (valueType) : description of value1
  key2 (keyType) : description of key2
  value2 (valueType) : description of value2
  ...
}
```

The key type and value type are optional. If one of the values is a sub-array, and the **array** type is specified, then the array specification may be nested as follows:

```
{
  key1 (keyType) : description of key1
  value1 (array) : description of value1
  {
    key2 (keyType) : description of key2
    value2 (valueType) : description of value2
    ...
  }
  ...
}
```

The indentation is not necessary, but is helpful for readability.

Any parameter specified using `<parameterName>` in the Syntax element should be described in the parameters section.

Value and return parameters

For simple return parameters, i.e. the direct return value of a function, use the Returns element. When a value is both an input and output value, for example the value of a property, use the Value element. The Returns and Value elements are mutually exclusive.

Some functions and commands in LiveCode change the value of 'the result' and the 'it' variable. In these cases there are also special return parameter elements It and The result.

All of these elements can have a type specified in brackets before the colon, as for parameter elements. The same comments about formatting apply to return parameters.

Description (required)

A full description of the API entry. The description may have markdown elements which will be rendered in the documentation viewer.

References

A comma-delimited list of other API or docs entries that are referenced. The references element is essential for linking around the documentation viewer.

The references list should be in the form:

entryName (entryType), entryName (entryType), entryName (entryType), ...

Tags

A comma-delimited list of tags for this entry. The documentation viewer can filter entries by tag.

Other documentation syntax

The Description element is the only one that allows standard markdown. There is only (essentially) one other syntactic form allowed in API documentation, which is enclosing text in angle brackets `<>`. There are three purposes of the angle brackets syntax:

- Specifying occurrences of the name of the current API entry.
- Specifying occurrences of the names of parameters.
- Specifying links to other API entries or documentation.

In the first two cases, applying the syntax merely consists in enclosing the relevant occurrences in angle brackets, eg to reference a parameter *pParam* in the entry named *thisEntry*,

This is a sentence in which `<pParam>` ought to be marked as being a parameter in the API entry for `<thisEntry>`

To specify a link to another API or docs entry, the latter **must** be included in the References element. If there is only one item in the References element with a matching name, then it suffices to enclose the referencing text in angle brackets.

This will link correctly to the `<other>` function's API entry provided the References element contains only one item with the name "other".

If the referencing text is different from the name of the entry, such as if it is pluralised or possessive, include the referenced entry name as a label inside the angle brackets.

This will also link correctly to `<other|other's>` API entry, but display as "other's" in the documentation viewer.

If the target name is shared by two references, then the link should be disambiguated by including the type in brackets:

We might want to link to the `<other(function)>` function and the `<other(command)>` command in the same entry, if the references contain "other (function)" and "other (command)".

It is possible to do both of the above at the same time:

This sentence contains links to `<other(function)|other's>` and `<other(command)|other's>` dictionary entries.

Separate docs files

To specify the documentation for a library in one separate file, it must have the following structure:

Library: *libraryName*

Summary: *librarySummary*

Description: *libraryDescription*

Name: *entryName*

Type: *entryType*

Syntax: *entrySyntax*

Summary: *entrySummary*

...other elements associated with *entryName*...

Name: *anotherEntryName* etc.

Inline documentation

Documentation for LiveCode script and LiveCode extensions can be done in-line. In this case, the Name, Type, Syntax and Associated elements are auto-generated.

The code-block enclosed in */** / immediately preceding the handler or syntax definition is used for the other elements of the documentation for that entry. In LiveCode extensions, the types of parameters are also pulled from the associated handler.

Separate docs files examples

Dictionary example

Here is the dictionary document for the *textEncode* function:

Name: `textEncode`

Type: `function`

Syntax: `textEncode(<stringToEncode>, <encoding>)`

Summary: `Converts from text to binary data.`

Introduced: `7.0`

OS: `mac,windows,linux,ios,android`

Platforms: `desktop,server,web,mobile`

Example:

`textEncode("A","UTF16")`

Example:

`put textEncode(field "output","UTF-8") into tOutput
put tOutput into url ("file:output.txt")`

Parameters:

`stringToEncode (string)`: Any string, or expression that evaluates to a string.
`encoding (enum)`: the encoding to be used

encoding (enum): the encoding to be used

- "ASCII"
- "ISO-8859-1": Linux only
- "MacRoman": OS X only
- "Native": ISO-8859-1 on Minux, MacRoman on OS X, CP1252 on Windows
- "UTF-16"
- "UTF-16BE"
- "UTF-16LE"
- "UTF-32"
- "UTF-32BE"
- "UTF-32LE"
- "UTF-8"
- "CP1252"

Returns: Returns the <stringToEncode> as binary data.

Description:

Converts from text to binary data.

The <textEncode> function takes text, and returns it as binary data, encoded with the specified encoding.

It is highly recommended that any time you interface with things outside LiveCode (files, network sockets, processes, etc) that you explicitly textEncode any text you send outside LiveCode and <textDecode> all text received into LiveCode. If this doesn't happen, a platform-dependent encoding will be used (which normally does not support Unicode text).

It is not, in general, possible to reliably auto-detect text encodings so please check the documentation for the programme you are communicating with to find out what it expects. If in doubt, try UTF-8.

References: textDecode (function)

Library example

Module: com.livecode.**sort**

Type: library

Description: This library consists of the sorting operations provided by the standard library of LiveCode Builder

Name: SortListDescendingText

Type: statement

Syntax: **sort** <Target> in descending [text] order

Summary: Sorts <Target> in descending text order.

Parameters:

Target (inout list): An expression that evaluates to a list of strings.

Example:

```
variable tTestList as List
put the empty list into tTestList

push "abcd" onto tTestList
push 1 onto tTestList
push "xyz" onto tTestList
push 2 onto tTestList

sort tTestList in descending order -- tTestList is ["xyz", "abcd", 1, 2]
```

Description: Text **sort** is performed by comparing string elements on a codepoint by codepoint basis. Non-string elements come after all string elements in the **sort** order. The **sort** is stable, so that non-string elements are not re-ordered relative to each other.

Tags: Sorting

Name: SortListAscendingText

Type: statement

Syntax: **sort** <Target> in ascending [text] order

Inline examples

In general, writing inline docs has fewer requirements since several of the elements are auto-generated.

LiveCode Builder syntax example

```

/**
This library consists of the operations on lists included in the standard library
of LiveCode Builder.
*/

module com.livecode.list

/**
Summary:      Returns the first element of <Target>.
Target:       An expression which evaluates to a list.
output:       The first element of <Target>

Example:
    variable tVar as List
    put the empty list into tVar
    push "first element" onto tVar

    variable tResult as Boolean
    if the head of tVar is "first element" then
        put "success" into tResult
    end if

Description:
Returns the first element of the list <Target> without modifying <Target>.
\\\'\'\'\' the head of tVar\\\'\'\'\'
is equivalent to using the <IndexedElementOfList> operator with index -1,
\\\'\'\'\' tVar[1]\\\'\'\'\'

References: IndexedElementOfList(operator)

Tags: Lists
*/

syntax HeadOfList is prefix operator with precedence 1
    "the" "head" "of" <Target: Expression>
begin
    MCListEvalHeadOf(Target, output)
end syntax

end module

```

LiveCode Builder handler example

```

/**
Summary: Logs the result of a test to the <xResults> list

Parameters:
pModule: The name of the module this test comes from.
pTest: The name of the test.
pResult: The result of the test
xResults: The ongoing list of test results

Description:
Pushes either the string "SUCCESS : <pModule>_<pTest>" or the string "FAILURE :
<pModule>_<pTest>" onto the results list,
depending on the value of <pResult>

*/

public handler testLog(in pModule as String, in pTest as String, in pResult as
Boolean, inout xResults as List)
    variable tStringResult as String
    if pResult then
        put "SUCCESS" into tStringResult
    else
        put "FAILURE" into tStringResult
    end if

    push tStringResult && ":" && pModule & "_" & pTest onto xResults
end handler

```

LiveCode script handler example

```

/**
Summary: Extracts the inline docs from a .lcb file

pFile: The path to the .lcb file to extract docs from

Returns (string): A string consisting of all the docs for the library, and the
syntax and handlers present in the .lcb file

Description:
<revDocsGenerateDocsFileFromModularFile> is used when packaging a widget to create
its API documentation.
It generates the Library and Type elements from the declaration in the <pFile>
(either widget or library), and extracts
the comment block that precedes any initial declaration for use as the library-
level Description element.
It then extracts the comment blocks that precede syntax and handler definitions in
<pFile>, and generates the
Name, Type, Syntax, and Associated elements for each entry, as well as the
parameter types.

Tags: Package building
*/

function revDocsGenerateDocsFileFromModularFile pFile
...
end revDocsGenerateDocsFileFromModularFile

```

LiveCode 9.6.4 Release Notes

Overview

LiveCode 9.0 enables access to libraries and platform APIs written in many other languages thanks to the community-funded 'Infinite LiveCode' project.

This includes a greatly improved LiveCode Builder virtual machine.

LiveCode 9.0 contains many additional improvements to support LiveCode app developers, including:

- A new "spinner" widget
- OAuth2 authentication library for use with web APIs (e.g. Facebook, Google and GitHub)
- A command argument parser library for building command-line standalones
- Updates and performance improvements for existing widgets

Known issues

- The installer will currently fail if you run it from a network share on Windows. Please copy the installer to a local disk before launching on this platform.
- The browser widget does not work on 32-bit Linux.
- 64-bit standalones for Mac OS X do not have support for audio recording.

Breaking changes

Boolean constants

In this release, boolean constants `true` and `false` have been changed so that they resolve to values of boolean type (rather than string). This will affect any uses of the `is strictly` operator on such values, i.e. previously the following were true:

`true is strictly a string` `false is strictly a string`

Now, they are both false, and the following are true:

`true is strictly a boolean` `false is strictly a boolean`

Boolean constants passed as elements of arrays to LCB handlers will not require conversion to boolean values in LCB - in fact any attempt to do so assuming they are strings will cause an error. Any array elements which are intended to be booleans in LCB should be checked for their type before conversion. For example, any of the following could be done by an LCB library user:

```
put true into tArray["enabled"]
put "true" into tArray["enabled"]
put (tVar is not "enabled") into tArray["enabled"]
```

An LCB handler to which `tArray` is passed should do the following:

```
variable tEnabled as Boolean
if tArray["enabled"] is a boolean then
    put tAction["enabled"] into tEnabled
else
    put tAction["enabled"] parsed as boolean into tEnabled
end if
```

Infinity constant

The constant `infinity` has been added to the language in this release. As a result, the unquoted literal `infinity` is now reserved. Any existing uses of it should be quoted, as otherwise it will resolve to the floating point value representing infinity, rather than the string "infinity".

Implicit object

A number of LCB commands use an implicit object to provide context for their execution. Some of these commands also allow specifying an explicit object. These commands are:

- `execute script`
- `send`
- `post`

- `image from file`
- `resolve file` - new in this version

In previous releases `execute script` and `image from file` would use `this card of the defaultStack` as the implicit object even if called from a widget. The `send` and `post` commands, however, used `this card of the defaultStack` when in a library module handler and the host widget when in a widget module handler. This release changes `execute script` and `image from file` to also use the host widget as the implicit object. This means, for example, that `image from file` will resolve a relative file path relative to the `stackFile` the host widget is on rather than the `stackFile` of the `defaultStack`.

Platform support

The engine supports a variety of operating systems and versions. This section describes the platforms that we ensure the engine runs on without issue (although in some cases with reduced functionality).

Windows

LiveCode supports the following versions of Windows:

- Windows 7 (both 32-bit and 64-bit)
- Windows Server 2008
- Windows 8.x (Desktop)
- Windows 10

Note: On 64-bit Windows installations, LiveCode can run either as a 32-bit application through the WoW layer or as a native 64-bit Windows application, depending on the installer that is chosen.

Linux

LiveCode supports the following Linux distributions, on 32-bit or 64-bit Intel/AMD or compatible processors:

- Ubuntu 14.04 and 16.04
- Fedora 23 & 24
- Debian 7 (Wheezy) and 8 (Jessie) [server]
- CentOS 7 [server]

LiveCode may also run on Linux installations which meet the following requirements:

- Required dependencies for core functionality:
 - glibc 2.13 or later
 - glib 2.0 or later
- Optional requirements for GUI functionality:
 - GTK/GDK 2.24 or later
 - Pango with Xft support
 - esd (optional, needed for audio output)
 - mplayer (optional, needed for media player functionality)
 - lcms (optional, required for color profile support in images)
 - gksu (optional, required for privilege elevation support)

Note: If the optional requirements are not present then LiveCode will still run but the specified features will be disabled.

Note: The requirements for GUI functionality are also required by Firefox and Chrome, so if your Linux distribution runs one of those, it will run LiveCode.

Note: It may be possible to compile and run LiveCode Community for Linux on other architectures but this is not officially supported.

Mac

The Mac engine supports:

- 10.9.x (Mavericks)
- 10.10.x (Yosemite)
- 10.11.x (El Capitan)
- 10.12.x (Sierra)
- 10.13.x (High Sierra)
- 10.14.x (Mojave)
- 10.15.x (Catalina)
- 11.x (Big Sur)

iOS

iOS deployment is possible when running LiveCode IDE on a Mac, and provided Xcode is installed and has been set in LiveCode *Preferences* (in the *Mobile Support* pane).

Currently, the supported versions of Xcode are:

- Xcode 9.2 on MacOS 10.12 (Note: You need to upgrade to 10.12.6)
- Xcode 10.1 on MacOS 10.13 (Note: You need to upgrade to 10.13.4)
- Xcode 11.3 on MacOS 10.14 (Note: You need to upgrade to 10.14.4)
- Xcode 12.4 on MacOS 10.15 and above (Note: You need to upgrade to 10.15.4)
- Xcode 12.5 on MacOS 11.0 and above

It is also possible to set other versions of Xcode, to allow testing on a wider range of iOS simulators. For instance, on MacOS 10.13 (High Sierra), you can add *Xcode 9.2* in the *Mobile Support* preferences, to let you test your stack on the *iOS Simulator 11.2*.

We currently support building against the following versions of the iOS SDK:

- 11.2 (included in Xcode 9.2)
- 12.1 (included in Xcode 10.1)
- 13.2 (included in Xcode 11.3)
- 14.4 (included in Xcode 12.4)
- 14.5 (included in Xcode 12.5)

Android

LiveCode allows you to save your stack as an Android application, and also to deploy it on an Android device or simulator from the IDE.

Android deployment is possible from Windows, Linux and Mac OSX.

The Android engine supports devices using x86, x86-64, ARM and ARM64 processors. It will run on the following versions of Android:

- 5.0-5.1 (Lollipop)
- 6.0 (Marshmallow)
- 7.x (Nougat)
- 8.x (Oreo)
- 9.0 (Pie)
- 10.0 (Q)

To enable deployment to Android devices, you need to download the [Android SDK](#), and then use the 'Android SDK Manager' to install:

- the latest "Android SDK Tools"
- the latest "Android SDK Platform Tools"

You also need to install the Java Development Kit (JDK). On Linux, this usually packaged as "openjdk". LiveCode requires JDK version 1.6 or later.

Once you have set the path of your Android SDK in the "Mobile Support" section of the LiveCode IDE's preferences, you can deploy your stack to Android devices.

Some users have reported successful Android Watch deployment, but it is not officially supported.

HTML5

LiveCode applications can be deployed to run in a web browser, by running the LiveCode engine in JavaScript and using modern HTML5 JavaScript APIs.

HTML5 deployment does not require any additional development tools to be installed.

LiveCode HTML5 standalone applications are currently supported for running in recent versions of [Mozilla Firefox](#), [Google Chrome](#) or [Safari](#). For more information, please see the "HTML5 Deployment" guide in the LiveCode IDE.

Setup

Installation

Each version of LiveCode installs can be installed to its own, separate folder. This allow multiple versions of LiveCode to be installed side-by-side. On Windows (and Linux), each version of LiveCode has its own Start Menu (or application menu) entry. On Mac OS X, each version has its own app bundle.

On Mac OS X, install LiveCode by mounting the `.dmg` file and dragging the app bundle to the `Applications` folder (or any other suitable location).

For Windows and Linux, the default installation locations when installing for "All Users" are:

Platform	Path
Windows	<code><x86 program files folder>/RunRev/LiveCode <version></code>
Linux	<code>/opt/livecode/livecode-<version></code>

The installations when installing for "This User" are:

Platform	Path
Windows	<code><user roaming app data folder>/RunRev/Components/LiveCode <version></code>
Linux	<code>~/.runrev/components/livecode-<version></code>

Note: If installing for "All Users" on Linux, either the **gksu** tool must be available, or you must manually run the LiveCode installer executable as root (e.g. using **sudo** or **su**).

Uninstallation

On Windows, the installer hooks into the standard Windows uninstall mechanism. This is accessible from the "Add or Remove Programs" applet in the windows Control Panel.

On Mac OS X, drag the app bundle to the Trash.

On Linux, LiveCode can be removed using the `setup.x86` or `setup.x86_64` program located in LiveCode's installation directory.

Reporting installer issues

If you find that the installer fails to work for you then please report it using the [LiveCode Quality Control Centre](#) or by emailing support@livecode.com.

Please include the following information in your report:

- Your platform and operating system version
- The location of your home or user folder
- The type of user account you are using (guest, restricted, admin etc.)
- The installer log file.

The installer log file can be located as follows:

Platform	Path
Windows 2000/XP	<documents and settings folder>/<user>/Local Settings/
Windows Vista/7	<users folder>/<user>/AppData/Local/RunRev/Logs
Linux	<home>/ .runrev/logs

Activating LiveCode

The licensing system ties your product licenses to a customer account system, meaning that you no longer have to worry about finding a license key after installing a new copy of LiveCode. Instead, you simply have to enter your email address and password that has been registered with our customer account system and your license key will be retrieved automatically.

Alternatively it is possible to activate the product via the use of a specially encrypted license file. These will be available for download from the customer center after logging into your account. This method will allow the product to be installed on machines that do not have access to the internet.

Command-line installation

It is possible to invoke the installer from the command-line on Linux and Windows. When doing command-line installation, no GUI will be displayed. The installation process is controlled by arguments passed to the installer.

Run the installer using a command in the form:

```
<installer> install -ui [OPTION ...]
```

where `<installer>` should be replaced with the path of the installer executable or app (inside the DMG) that has been downloaded. The result of the installation operation will be written to the console.

The installer understands any of the following `OPTION`s:

Option	Description
<code>-allusers</code>	Install the IDE for "All Users". If not specified, LiveCode will be installed for the current user only.
<code>-desktopshortcut</code>	Place a shortcut on the Desktop (Windows-only)
<code>-startmenu</code>	Place shortcuts in the Start Menu (Windows-only)
<code>-location LOCATION</code>	The folder to install into. If not specified, the <code>LOCATION</code> defaults to those described in the "Installation" section above.
<code>-log LOGFILE</code>	The file to which to log installation actions. If not specified, no log is generated.

Note: the command-line installer does not do any authentication. When installing for "All Users", you will need to run the installer command as an administrator.

As the installer is actually a GUI application, it needs to be run slightly differently from other command-line programs.

On Windows, the command is:

```
start /wait <installer> install -ui [OPTION ...]
```

Command-line uninstallation

It is possible to uninstall LiveCode from the command-line on Windows and Linux. When doing command-line uninstallation, no GUI will be displayed.

Run the uninstaller using a command of the form:

```
<uninstaller> uninstall -ui
```

Where is *.setup.exe* on Windows, and *.setup.x86* on Linux. This executable, for both of the platforms, is located in the folder where LiveCode is installed.

The result of the uninstallation operation will be written to the console.

Note: the command-line uninstaller does not do any authentication. When removing a version of LiveCode installed for "All Users", you will need to run the uninstaller command as an administrator.

Command-line activation

It is possible to activate an installation of LiveCode for all users by using the command-line. When performing command-line activation, no GUI is displayed. Activation is controlled by passing command-line arguments to LiveCode.

Activate LiveCode using a command of the form:

```
<livecode> activate -file LICENSEFILE -passphrase SECRET
```

where `<livecode>` should be replaced with the path to the LiveCode executable or app that has been previously installed.

This loads license information from the manual activation file `LICENSEFILE`, decrypts it using the given `SECRET` passphrase, and installs a license file for all users of the computer. Manual activation files can be downloaded from the [My Products](#) page in the LiveCode account management site.

It is also possible to deactivate LiveCode with:

```
<livecode> deactivate
```

Since LiveCode is actually a GUI application, it needs to be run slightly differently from other command-line programs.

On Windows, the command is:

```
start /wait <livecode> activate -file LICENSE -passphrase SECRET  
start /wait <livecode> deactivate
```

On Mac OS X, you need to do:

```
<livecode>/Contents/MacOS/LiveCode activate -file LICENSE -passphrase SECRET  
<livecode>/Contents/MacOS/LiveCode deactivate
```

Previous release notes