LiveCode 6.7.0-dp-1 Release Notes

Table of contents

Overview

This document describes all the changes that have been made for LiveCode 6.7.0-dp-1, including bug fixes and new syntax.

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.

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

The engine supports the following Windows OSes:

- 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
- Windows 8.x (Desktop)

Note: *On 64-bit platforms the engine still runs as a 32-bit application through the WoW layer.*

Linux

The linux engine requires the following:

- 32-bit installation, or a 64-bit linux distribution that has a 32-bit compatibility layer
- 2.4.x or later kernel
- X11R5 capable Xserver running locally on a 24-bit display
- glibc 2.3.2 or later
- 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)

Note: *The optional requirements (except for gksu and lcms) are also required by Firefox and Chrome, so if your linux distribution runs one of those, it will run the engine.*
Note: *If the optional requirements are not present then the engine will still run but the specified features will be disabled.*
Note: *LiveCode and standalones it builds may work on remote Xservers and in other bit-depths, however this mode of operation is not currently supported.*

Mac

The Mac engine supports:

- 10.5.8 and later (Leopard) on Intel and PowerPC
- 10.6.x (Snow Leopard) on Intel
- 10.7.x (Lion) on Intel
- 10.8.x (Mountain Lion) on Intel
- 10.9.x (Mavericks) on Intel

Note: *The engine runs as a 32-bit application regardless of the capabilities of the underlying processor.*

Setup

Installation

Each distinct version has its own complete folder – multiple versions will no longer install side-by-side: on Windows (and Linux), each distinct version will gain its own start menu (application menu) entry; on Mac, each distinct version will have its own app bundle.
The default location for the install on the different platforms when installing for 'all users' are:

- Windows: <x86 program files folder>/RunRev/ LiveCode 6.7.0-dp-1
- Linux: /opt/runrev/livecode-6.7.0-dp-1
- Mac: /Applications/ LiveCode 6.7.0-dp-1.app

The default location for the install on the different platforms when installing for 'this user' are:

- Windows: <user roaming app data folder>/RunRev/Components/LiveCode 6.7.0-dp-1
- Linux: ~/.runrev/components/livecode-6.7.0-dp-1
- Mac: ~/Applications/ LiveCode 6.7.0-dp-1.app

Note: *If your linux distribution does not have the necessary support for authentication (gksu) then the installer will run without admin privileges so you will have to manually run it from an admin account to install into a privileged location.*

Uninstallation

On Windows, the installer hooks into the standard Windows uninstall mechanism. This is accessible from the appropriate pane in the control panel.
On Mac, simply drag the app bundle to the Trash.
On Linux, the situation is currently less than ideal:

- open a terminal
- *cd* to the folder containing your rev install. e.g.

```
cd /opt/runrev/livecode-6.7.0-dp-1
```

- execute the *.setup.x86* file. i.e.

```
./.setup.x86
```

- follow the on-screen instructions.

Reporting installer issues

If you find that the installer fails to work for you then please file a bug report in the RQCC or email support@runrev.com so we can look into the problem.
In the case of failed install it is vitally important that you include the following information:

- Your platform and operating system version
- The location of your home/user folder
- The type of user account you are using (guest, restricted, admin etc.)
- The installer log file located as follows:
- **Windows 2000/XP:** <documents and settings folder>/<user>/Local Settings/

- **Windows Vista/7:** <users folder>/<user>/AppData/Local/RunRev/Logs
- **Linux:** <home>/.runrev/logs
- **Mac:** <home>/Library/Application Support/Logs/RunRev

Activation

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.

Multi-user and network install support (4.5.3)

In order to better support institutions needing to both deploy the IDE to many machines and to license them for all users on a given machine, a number of facilities have been added which are accessible by using the command-line.
Note: *These features are intended for use by IT administrators for the purposes of deploying LiveCode in multi-user situations. They are not supported for general use.*

Command-line installation

It is possible to invoke the installer from the command-line on both Mac and Windows. When invoked in this fashion, no GUI will be displayed, configuration being supplied by arguments passed to the installer.
On both platforms, the command is of the following form:

<exe> install noui *options*

Here *options* is optional and consists of one or more of the following:

-allusers	Install the IDE for all users. If not specified, the install will be done for the current user only.
-desktopshortcut	Place a shortcut on the Desktop (Windows-only)
-startmenu	Place shortcuts in the Start Menu (Windows-only)
-location *location*	The location to install into. If not specified, the location defaults to those described in the *Layout* section above.
-log *logfile*	A file to place a log of all actions in. If not specified, no log is generated.

Note that the command-line variant of the installer does not do any authentication. Thus, if you wish to install to an admin-only location you will need to be running as administrator before executing the command.
As the installer is actually a GUI application, it needs to be run slightly differently from other command-line programs.
In what follows <installerexe> should be replaced with the path of the installer executable or app (inside the DMG) that has been downloaded.
On Windows, you need to do:

start /wait <installerexe> install noui *options*

On Mac, you need to do:

"<installerexe>/Contents/MacOS/installer" install noui *options*

On both platforms, the result of the installation will be written to the console.

Command-line activation

In a similar vein to installation, it is possible to activate an installation of LiveCode for all-users of that machine by using the command-line. When invoked in this fashion, no GUI will be displayed, activation being controlled by any arguments passed.
On both platforms, the command is of the form:

<exe> activate -file *license* -passphrase *phrase*

This command will load the manual activation file from *license*, decrypt it using the given *passphrase* and then install a license file for all users of the computer. Manual activation files can be downloaded from the 'My Products' section of the RunRev customer accounts area.
This action can be undone using the following command:

<exe> deactivate

Again, as the LiveCode executable is actually a GUI application it needs to be run slightly differently from other command-line programs.
In what follows <livecodeexe> should be replaced with the path to the installed LiveCode executable or app that has been previously installed.
On Windows, you need to do:

start /wait <livecodeexe> activate -file *license* -passphrase *phrase*
start /wait <livecodeexe> deactivate

On Mac, you need to do:

"<livecodeexe>/Contents/MacOS/LiveCode" activate -file *license* -passphrase *phrase*
"<livecodeexe>/Contents/MacOS/LiveCode" deactivate

On both platforms, the result of the activation will be written to the console.

Proposed changes

The following changes are likely to occur in the next or subsequent non-maintenance release:

- The engine (both IDE and standalone) **will require** gtk, gdk, glib, pango and xft on Linux

Engine changes

Clipboard data 'styledText' array accessor. (6.7.0-dp-1)

A new clipboard format has been added 'styledText'. This format returns (or sets) the clipboard to a styled text array - the same format as the 'styledText' property of field chunks. All text formats can convert to and from the 'styledText' key.
For example, you can now do:
set the clipboardData["styledText"] to the styledText of line 5 of field 3
set the styledText of line 6 of field 3 to the clipboardData["styledText"]

Note that the dragData can now also be used with this new format in exactly the same way.

Cocoa Support (6.7.0-dp-1)

With 6.7 we have replaced the majority of Carbon API usage with Cocoa. The goals of this work are three-fold:

- Allow embedding of native 'NSViews' into LiveCode windows (in particular, browser controls).
- Enable submission of LiveCode apps to the Mac AppStore.
- Enable eventual building of 64-bit versions of LiveCode for Mac.

As of DP 1 we have achieved the first goal and revBrowser has been updated as a result. The main upshot of this is that the browser is now part of the host window and as such works correctly regardless of the type of window (dialog, palette, document etc).

The instability issues caused by the AppStore sandbox when using mixed Cocoa and Carbon APIs should also be resolved in DP 1. However, the engine still statically links with QuickTime and QTKit which are now not allowed in apps submitted to the Mac AppStore. This will be addressed in DP 2 along with an AVKit implementation of the player and other multimedia features.

The final goal (64-bit support) will be gradually worked towards over the next few LiveCode versions as the engine gets 'decarbonated' (usage of Carbon APIs which do not have 64-bit equivalents removed).

As there has been quite a substantial rework on the Mac port it is expected that there will be issues to address during the release cycle. We want to ensure that the functionality of 6.7 is as close as possible to that of 6.6, so please do report any differences you notice however minor you think they might be.

With the release of dp-1 there are a number of known issues:

- No backdrop support - we are currently working out how to implement this feature using Cocoa APIs
- No drawer support - we are currently working out how to implement this feature using Cocoa APIs
- Cursor issues over window borders and during drag-drop - the cursor will sometimes stick or change to the wrong type, this is being investigated.
- Cmd-Shift-'_' does not work - this is being investigated.
- Themed scrollbars sometimes do not work correctly on Retina displays - this is being investigated.
- Some aspects of the player are currently non-functional - in particular, user callbacks and QTVR related properties.

Finally, an important internal change which will affect maintainers of Mac externals that use the windowId is that this property now returns the 'global window number' (which is the unique ID the Window Server uses to identify windows). To turn this into a Cocoa NSWindow pointer use [NSApp windowWithWindowNumber: t_window_id]. Note that it is no longer possible to get a Carbon WindowRef, nor should this be attempted as

trying to mix Carbon and Cocoa in this manner will cause instability inside the sandbox environment required by the Mac AppStore.

Improved revBrowser external (6.7.0-dp-1)

The revBrowser external has been updated to support Cocoa on OSX, and now embeds the browser control properly within the window.
In addition a new browser component based on CEF (Chromium Embedded Framework) has been added.

This new browser allows for a consistent appearance across all platforms with a modern, well supported feature set.

To use the new CEF browser use the *revBrowserOpenCef* command in place of *revBrowserOpen*. This will create a CEF browser instance which can be used with the existing revBrowser commands and functions in exactly the same way as before.

JavaScript integration

The new chrome browser allows us to add the ability to call LiveCode handlers from within the browser using JavaScript. To make a LiveCode handler visible to JavaScript, use the *revBrowserAddJavaScriptHandler* command, and to remove it use the *revBrowserRemoveJavaScriptHandler* command. LiveCode handlers are added as functions with the same name attached to a global 'liveCode' object. When called, these functions will result in the corresponding LiveCode handler message being sent to the browser card with the browser instance ID and any function arguments as parameters.

Example:

With the handler "myJSHandler" registered using *revBrowserAddJavaScriptHandler*, it can be called from the browser like so:

liveCode.myJSHandler(tFieldContents, tAction);

the LiveCode handler would then be called with the following parameters:

- pBrowserInstance (the browser instance id, as returned from the *revOpenBrowserCef* function)
- pFieldContents (the first argument of the JavaScript function call)
- pAction (the second argument of the JavaScript function call)

In-App Purchasing (6.7.0-dp-1)

Why has the API changed?

The LiveCode engine until now supported in-app purchasing for apps distributed through the Google Play store (formerly Android Market), as well as the Apple AppStore. This support is now extended so that apps distributed through other avenues (the Amazon & Samsung app stores) can make use of the in-app purchase features provided. For this reason, new LiveCode commands have been added, and some of the old ones have slightly changed. However, all of the old commands are still supported (for the Google Play Store and the Apple AppStore). In order the existing scripts users have written to continue to work, all it needs is to add one or two extra lines, depending on the store. More details on this later. Moreover, the new API allows the user to query specific product information (such as price, description etc) before they make a purchase, and supports purchasing of subscription items for all available stores. Furthermore, for the Google Play Store, the new API uses the newest version of Google In-App Billing API (v3), that offers synchronous purchase flow, and purchase information is available immediately after it completes. This information of in-app purchases is maintained within the Google Play system until the purchase is *consumed*. More on the

consumption of purchased items later.

What has changed?

To start with, the main changes are the following:

- Each item has an extra property, the *itemType*, that has to be specified before making a purchase. This is done using the **mobileStoreSetProductType** command. The *itemType* can either be *subs*, for subscription items, or *inapp* for consumable and non-consumable items.

- Due to a restriction of the newest version of Google In-App Billing API, you cannot buy consumable items more than once, unless you consume them. This is done using the **mobileStoreConsumePurchase** command. Note that this command is actually only used when interacting with the Google Play Store API. What it does is sending a consumption request to Google Play, so that you will be able to buy this product again. You would typically implement consumption for items that can be purchased multiple times (i.e. for consumable products, such as in-game currency, fuel etc). Note that in case you call mobileStoreConsumePurchase on a non consumable product, then you no longer own this item.

- The new purchase flow has become simpler.

  Instead of

  - creating a purchase request (**mobilePurchaseCreate** productID)

  - store the new purchase request ID (put the result into tPurchaseID),

  - setting properties such as quantity and developer payload (**mobilePurchaseSet** tPurchaseID, "quantity", pQuantity)

  - sending a purchase request to the store (**mobilePurchaseSendRequest** tPurchaseID)

  now all it needs is just

  - set the product type (**mobileStoreSetProductType** productID, itemType)

  - make a purchase (**mobileStoreMakePurchase** productID, quantity, developerPayload)

- The **purchaseStateUpdate** message that the store sends in response to **mobileStoreMakePurchase**, contains not only the purchase identifier and the state of the purchase, but also the product identifier of the requested item:

  purchaseStateUpdate *purchaseID*, *productID*, *state*

- So you can query a purchased product property using the product identifier, instead of the purchase identifier:

  mobileStoreProductProperty *productID*, *propertyName*

  Note that the old function **mobileGetPurchase** *purchaseID*, *propertyName* will still work.

- You can get information on a specific item (such as product identifier, product type, price etc), using the **mobileStoreRequestProductDetails** command. The store responds:

  In case the request is successful, a **productDetailsReceived** message is sent by the store.

  In case of failure, a **productRequestError** message is sent by the store.

- You can get a list of all known completed purchases using **mobileStorePurchasedProducts** function. This returns a list of product identifiers of restored or newly bought purchases.

What needs to change in existing scripts?

It is recommended that scripts which were written using previous versions of LiveCode (and thus use the old LiveCode API for in-app purchasing), should be used to run on these versions. However, it is still possible to run an existing script (that makes use of in-app purchasing feature) on LiveCode 6.7, only by changing a few things:

- **purchaseStateUpdate** message is now called with 3 parameters, (purchaseID, productID, state), instead of two (purchaseID, state). This applies to apps built for both the Google Play Store and the Apple AppStore.

- before sending a **mobilePurchaseSendRequest**, you have to specify the type (*subs* or *inapp*) of the item using **mobileStoreSetProductType** *productID*, *type* command (Google Play Store only).

- if you want to buy more than one consumable item, you have to consume it first. This can be done by using the **mobileStoreConsumePurchase** *productID* command (Google Play Store only).

If you want to build apps for Amazon and/or Samsung Store, you have to use the newest LiveCode API. Â

How to use the new API?

Setup

Before you can use IAP, you must set up products in each vendor's developer portal. In brief, you have to:

- Create each product you want to sell, giving it a unique identifier. Note that for the Samsung Seller Office, the developer cannot choose the product identifier. This is assigned by the store.
- Submit the items for approval to the appropriate store. Some stores may require additional metadata, such as screenshots of your for sale items.
- Set up unique test accounts. The user is not charged when making a purchase using the test account details. This applies to Apple and Google. Amazon and Samsung have different methods for testing.

For more detailed store-specific information, you can have a look at the links below:

Apple AppStore

Google Play Store

Amazon Appstore

Samsung Apps Store and more specifically click here

Purchase Types

There are three classes of products users can purchase:

1. One-time purchases that get "consumed". Typically, these items are called *consumables*. The user can buy as many times as they want (virtual coins/bullets in a game), except in apps built for the Google Play Store, where the user has to consume the purchased item first, and then buy (one) more.
2. One-time purchases that last forever, such as unlocking extra features, downloading new content once. These items are usually called *non-consumables*.
3. Subscriptions where the app user pays a periodical fee to receive some ongoing service. Subscriptions can either be auto-renewable or non-renewable.

Each vendor uses different terminology for these purchases :

	Apple	Google	Amazon	Samsung
one-time, gets consumed	consumable	unmanaged	consumable	consumable
one-time, lasts forever	non-consumable	managed	entitlement	non-consumable
subscriptions	auto-renewable , non-renewable	auto-renewable	auto-renewable	non-renewable

Testing

Again, each store uses a different method of testing.

For the Apple AppStore, you can create test accounts. More details here.

For the Google Play Store, you can create test accounts as well as test using static responses. More details here. Note that you cannot test subscriptions using the test account. This means that the test user will be charged when purchasing a subscription item. A possible workaround to this, is to log into the Google Wallet Service as a seller, using your Google Developer account details, and "refund" and then "cancel" the order of the subscription item that the test user had just purchased.

For the Amazon Appstore, you can test your app using SDK Tester. This is a developer tool that allows users of the Amazon Mobile App SDK to test their implementation in a production-like environment before submitting it to Amazon for publication. More details here.

For the Samsung Apps Store, Samsung IAP API offers three modes to test the service under various conditions : *Production Mode, Test Mode Success, Test Mode Fail*. During development period, you can select the mode in the Standalone Application Settings window. Before releasing your application, you must change to Production Mode. If you release your application in Test Mode, actual payments will not occur. More details on page 6 and 7 here.

Note that in Production Mode, your app can only interact with item groups with *sales* status. This information exists in the Samsung Seller Office. However, item groups are only given sales status after the app has been certified. In other words, you can test your app in Production Mode only after it has been certified by Samsung.

Syntax

Implementing in-app purchasing requires two way communication between your LiveCode app and the vendor's store. Here is the basic process:

- Your app sends a request to purchase a specific in-app purchase to the store
- The store verifies this and attempts to take payment
- If payment is successful the store notifies your app
- Your app unlocks features or downloads new content / fulfils the in-app purchase
- Your app tells the store that all actions associated with the purchase have been completed
- Store logs that in-app purchase has been completed

Commands, Functions and Messages

To determine if in-app purchasing is available use:

mobileStoreCanMakePurchase()

Returns *true* if in-app purchases can be made, *false* if not.

Throughout the purchase process, the store sends **purchaseStateUpdate** messages to your app which report any changes in the status of active purchases. The receipt of these messages can be switched on and off using:

mobileStoreEnablePurchaseUpdates
mobileStoreDisablePurchaseUpdates

If you want to get information on a specific item (such as product identifier, product type, price etc), you can use:

mobileStoreRequestProductDetails *productID*

The *productID* is the identifier of the item you are interested. Then, the store sends a *productDetailsReceived* message, in case the request is successful, otherwise it sends a *productRequestError* message:

productDetailsReceived *productID*, *details*

The *productID* is the identifier of the item, and *details* is an array with the following keys - that are different depending on the store:

For Android stores (Google, Amazon, Samsung), the keys are:

- *productID* : identifier of the requested product
- *price* : price of the requested product
- *description* : description of the requested product
- *title* : title of the requested product
- *itemType* : type of the requested product
- *itemImageUrl* : URL where the image (if any) of the requested product is stored
- *itemDownloadUrl* : URL to download the requested product
- *subscriptionDurationUnit* : subscription duration unit of the requested product
- *subscriptionDurationMultiplier* : subscription duration multiplier of the requested product

Note that some Android stores do not provide values for all the above keys. In this case, the value for the corresponding key will be empty.

For iTunes Connect store (Apple), the keys of *details* array are the following:

- *price* : price of the requested product
- *description* : description of the requested product
- *title* : title of the requested product
- *currency code* : price currency code of the requested product
- *currency symbol* : currency symbol of the requested product
- *unicode description* : unicode description of the requested product
- *unicode title* : unicode title of the requested product
- *unicode currency symbol* : unicode currency symbol of the requested product

If **mobileStoreRequestProductDetails** is not successful, then a *productRequestError* message is sent :

productRequestError *productID*, *error*

The *productID* is the identifier of the item, and *error* is a string that describes the error.

Before sending a purchase request for a particular item, you have to specify the type of this item. To do this, use :

mobileStoreSetProductType *itemType*

The *itemType* can either be *subs* or *inapp*.

To create and send a request for a new purchase use:

mobileStoreMakePurchase *productID*, *quantity*, *developerPayload*

The *productID* is the identifier of the in-app purchase you created in the vendor's developer portal and wish to purchase. The *quantity* specifies the quantity of the in-app purchase to buy (iOS only - always "1" in Android) . The *developerPayload* is a string of less than 256 characters that will be returned with the purchase details once complete. Can be used to later identify a purchase response to a specific request (Android only).

To get a list of all known completed purchases use:

mobileStorePurchasedProducts()

It returns a return-separated list of product identifiers, of restored or newly bought purchases which are confirmed as complete. Note that in iOS, consumable products as well as non-renewable subscriptions will not be contained in this list.

Once a purchase is complete, you can retrieve the properties of the purchased product, using:

mobileStoreProductProperty *(productID, property)*

The parameters are as follows:

- *productID* : identifier of the requested product
- *property* : name of the purchase request property to get

Properties which can be queried can differ depending on the store:

For the Samsung Apps Store (Android), you can query the properties:

- *title* : title of the purchased product
- *productId* : identifier of the purchased product
- *price* : price of the purchased product
- *currencyUnit* : currency unit of the product price
- *description* : description of the product as specified in the Samsung Seller Office
- *itemImageUrl* : URL where the image of the purchased product is stored
- *itemDownLoadUrl* : URL to download the purchased product
- *paymentId* : payment identifier of the purchased product
- *purchaseId* : purchase identifier of the purchased product
- *purchaseDate* : purchase date, in milliseconds
- *verifyUrl* : IAP server URL for checking if the purchase is valid for the IAP server, using the *purchaseId* value

For the Google Play Store (Android), you can query the properties:

- *productId* : identifier of the purchased product
- *packageName* : application package from which the purchase originated
- *orderId* : unique order identifier for the transaction. This corresponds to the Google Wallet Order ID
- *purchaseTime* : time the product was purchased, in milliseconds
- *developerPayload* : developer-specified string that contains supplemental information about an order. You can specify a value for this in **mobileStoreMakePurchase**
- *purchaseToken* : token that uniquely identifies a purchase for a given item and user pair.

- *itemType* : type of the purchased item, *inapp* or *subs*
- *signature* : string containing the signature of the purchase data that was signed with the private key of the developer. The data signature uses the RSASSA-PKCS1-v1_5 scheme

For the Amazon Appstore (Android), you can query the properties:

- *productId* : identifier of the purchased product
- *itemType* : type of the purchased product. This can be *CONSUMABLE*, *ENTITLED* or *SUBSCRIPTION*
- *subscriptionPeriod* : string indicating the start and end date for subscription (for subscription products only)
- *purchaseToken* : purchase token that can be used from an external server to validate purchase

For Apple AppStore (iOS), you can query the properties:

- *quantity* : amount of item purchased. You can specify a value for this in **mobileStoreMakePurchase**
- *productId* : identifier of the purchased product
- *receipt* : block of data that can be used to confirm the purchase from a remote server with the iTunes Connect store
- *purchaseDate* : date the purchase / restoration request was sent
- *transactionIdentifier* : unique identifier for a successful purchase / restoration request
- *originalPurchaseDate* : date of the original purchase, for restored purchases
- *originalTransactionIdentifier* : the transaction identifier of the original purchase, for restored purchases
- *originalReceipt* : the receipt for the original purchase, for restored purchases

Once you have sent your purchase request and it has been confirmed, you can then unlock or download new content to fulfil the requirements of the in-app purchase. You must inform the store once you have completely fulfiled the purchase using:

mobileStoreConfirmPurchase *productID*

Here, *productID* is the identifier of the product requested for purchase.

mobileStoreConfirmPurchase should only be called on a purchase request in the *paymentReceived* or *restored* state (more on the states of the purchase later). If you don't send this confirmation before the app is closed, **purchaseStateUpdate** messages for the purchase will be sent to your app the next time updates are enabled by calling the **mobileStoreEnablePurchaseUpdates** command.

To consume a purchased product use:

mobileStoreConsumePurchase *productID*

Here, *productID* is the identifier of the product requested for consumption. Note that this command is actually only used when interacting with the Google Play Store API. This is because the Google Play Store API has a restriction that ensures a consumable product is consumed before another instance is purchased. *Consume* means that the purchase is removed from the user's inventory of purchased items, allowing the user buy that product again.

Note that **mobileStoreConsumePurchase** must only be called on consumable products. If you call **mobileStoreConsumePurchase** on a non-consumable product, then you no longer own this product.

To instruct the store to re-send notifications of previously completed purchases use:

mobileStoreRestorePurchases

This would typically be called the first time an app is run after installation on a new device to restore any items bought through the app.

To get more detailed information about errors in the purchase request use:

mobileStorePurchaseError *(purchaseID)*

The store sends **purchaseStateUpdate** messages to notifies your app of any changes in state to the purchase request. These messages continue until you notify the store that the purchase is complete or it is cancelled.

purchaseStateUpdate *purchaseID, productID, state*

The state can be any one of the following:

- *sendingRequest* : the purchase request is being sent to the store / marketplace
- *paymentReceived* : the requested item has been paid for. The item should now be delivered to the user and confirmed via the mobileStoreConfirmPurchase command
- *alreadyEntitled* : the requested item is already owned, and cannot be purchased again
- *invalidSKU* : the requested item does not exist in the store listing
- *complete* : the purchase has now been paid for and delivered
- *restored* : the purchase has been restored after a call to mobileStoreRestorePurchases. The purchase should now be delivered to the user and confirmed via the mobileStoreConfirmPurchase command
- *cancelled* : the purchase was cancelled by the user before payment was received
- *error* : An error occurred during the payment request. More detailed information is available from the mobileStorePurchaseError function

OS 10.5 (Leopard) Support (6.7.0-dp-1)

As of version 6.7-dp-1, Mac OS 10.5 (Leopard) support has been dropped from LiveCode. This is primarily for technical reasons: In order to support the latest OS X features (e.g. Cocoa) dropping 10.5 support was required.

As Leopard was the last Mac version to support PPC, support for the PPC architecture has also been dropped and the Universal and PPC options have been removed from the Standalone Builder.

Users wishing to produce 10.5 compatible executables can still do so using LiveCode version 6.6.x (and earlier).

Setting the label of an option or combo-box does not update the menuHistory. (6.7.0-dp-1)

Previously, setting the label of an option or combo-box control would not update the menuHistory property. Now, setting the label of such a control will search through the list of items in the control and set the menuHistory to the first item that matches (taking into account the setting of the caseSensitive local property).
Note: Unlike setting the menuHistory property direct, this does not cause a menuPick message to be sent.

pixelScaling not enabled on Windows Commercial edition (6.7.0-dp-1)

Specific bug fixes (6.7.0-dp-1)

(bug fixes specific to the current build are highlighted in bold, reverted bug fixes are stricken through)

12010 Windows engine hangs after multiple stack redraws.
11975 "import snapshot from rect ..." only imports part of the screen on Windows
11946 iOS 7.1 Simulator doesn't remember device type when launching using 'Test'
11917 Setting the label of an option or combo-box does not update the menuHistory.
11808 pixelScaling not enabled on Windows Commercial edition

Dictionary additions

- **mobileStoreConfirmPurchase** (*command*) has been added to the dictionary.
- **mobileStoreConsumePurchase** (*command*) has been added to the dictionary.
- **mobileStoreDisablePurchaseUpdates** (*command*) has been added to the dictionary.
- **mobileStoreEnablePurchaseUpdates** (*command*) has been added to the dictionary.
- **mobileStoreMakePurchase** (*command*) has been added to the dictionary.
- **mobileStoreRequestProductDetails** (*command*) has been added to the dictionary.
- **mobileStoreRestorePurchases** (*command*) has been added to the dictionary.
- **mobileStoreSetProductType** (*command*) has been added to the dictionary.
- **mobileStoreVerifyPurchase** (*command*) has been added to the dictionary.
- **revBrowserAddJavaScriptHandler** (*function*) has been added to the dictionary.
- **revBrowserRemoveJavaScriptHandler** (*function*) has been added to the dictionary.
- **mobileStoreCanMakePurchase** (*function*) has been added to the dictionary.
- **mobileStoreProductProperty** (*function*) has been added to the dictionary.
- **mobileStorePurchaseError** (*function*) has been added to the dictionary.
- **mobileStorePurchasedProducts** (*function*) has been added to the dictionary.
- **revBrowserOpenCef** (*function*) has been added to the dictionary.
- **productDetailsReceived** (*message*) has been added to the dictionary.
- **productRequestError** (*message*) has been added to the dictionary.
- **purchaseStateUpdate** (*message*) has been added to the dictionary.

Previous Release Notes

6.6.0 Release Notes	http://downloads.livecode.com/livecode/6_6_0/LiveCodeNotes-6_6_0.pdf
6.5.2 Release Notes	http://downloads.livecode.com/livecode/6_5_2/LiveCodeNotes-6_5_2.pdf
6.5.1 Release Notes	http://downloads.livecode.com/livecode/6_5_1/LiveCodeNotes-6_5_1.pdf
6.5.0 Release Notes	http://downloads.livecode.com/livecode/6_5_0/LiveCodeNotes-6_5_0.pdf
6.1.3 Release Notes	http://downloads.livecode.com/livecode/6_1_3/LiveCodeNotes-6_1_3.pdf
6.1.2 Release Notes	http://downloads.livecode.com/livecode/6_1_2/LiveCodeNotes-6_1_2.pdf
6.1.1 Release Notes	http://downloads.livecode.com/livecode/6_1_1/LiveCodeNotes-6_1_1.pdf
6.1.0 Release Notes	http://downloads.livecode.com/livecode/6_1_0/LiveCodeNotes-6_1_0.pdf
6.0.2 Release Notes	http://downloads.livecode.com/livecode/6_0_2/LiveCodeNotes-6_0_2.pdf
6.0.1 Release Notes	http://downloads.livecode.com/livecode/6_0_1/LiveCodeNotes-6_0_1.pdf
6.0.0 Release Notes	http://downloads.livecode.com/livecode/6_0_0/LiveCodeNotes-6_0_0.pdf