Android Deployment Release Notes

Table of Contents

Overview ... 3
Getting Started ... 4
  Prerequisites ... 4
  Configuring LiveCode ... 4
  Configuring an Android standalone ... 5
  Configuring an emulated device ... 5
  Configuring a real device ... 6
  Testing an Android application ... 6
  A First Project ... 7
Configuring an Android Application ... 8
  Setting manifest options ... 8
  Adding a launcher icon ... 9
  Adding a splash image (personal and educational) ... 9
  Adding a default launch image (trial) ... 10
Deployment Features ... 11
  Standalone builder messages ... 11
General Engine Features ... 12
  Engine version ... 12
  What doesn't work ... 12
  What does work ... 12
  Windowing and Stacks ... 13
  System Dialogs – answer and ask ... 13
  Non-file URL access ... 13
  Externals ... 15
Android Specific Engine Features ... 16
  Limitations ... 16
  Multi-touch events ... 16
  Mouse events ... 16
  Motion events ... 17
  Hardware button support ... 17
  Accelerometer support ... 17
  Photo album and camera support ... 18
    Taking or choosing photos ... 18
  Keyboard Input ... 18
    Configuring keyboard type ... 18
  Orientation handling ... 19
    Auto-rotation support ... 19
    Querying orientation ... 19
    Controlling auto-rotation ... 20
    Orientation changed notification ... 20
    Device specific orientations ... 20
  Resolution handling ... 21
  Email composition ... 22

Basic support....22
Advanced support....22
File and folder handling....23
Basic sound playback support....24
Video playback support....25
URL launching support....25
Hardware and system version query support....26
Querying camera capabilities....26
Status bar configuration support....27
Device Information....27
In App Purchasing....28
Syntax....28
Commands & Functions....28
Messages....30
Noteworthy Changes....31
OpenGL Compositor (5.0.1-dp-1)....31
Change Logs and History....32
Engine Change History....32
Deployment Change History....34
Document Change History....35

Overview

LiveCode now incorporates facilities for deploying to Android. These facilities include the ability to build Android applications that run in the Android emulator as well as on Android devices.

In addition to supporting much of the desktop engine's features, the Android engine hooks into many Android-specific features. Please see the *Android Specific Features* section for more details.

For information on what parts of the Desktop feature set are currently implemented when deploying to Android, please see the *What Works* section.

Note: *If you have not purchased the Android deployment pack, you can still try out Android deployment features, but any built apps will have a forced banner for 5 seconds on startup, and will quit after one minute.*

Note: *Android deployment is only supported on Windows and Mac machines.*

Important: *The LiveCode Android engine supports Android devices running version 2.2 and higher only.*

Getting Started

Before you can use the Android plugin, you need to ensure you have set up your system appropriately.

This section contains all the salient details you need to setup your system, but there are step-by-step lessons taking you through this in more detail for each platform here:

http://www.runrev.com/links/setup-android-mac

http://www.runrev.com/links/setup-android-windows

Note: *The above lessons are currently being updated to take account of the integration of the Android plugin facilities into the IDE.*

Prerequisites

If you are intending to use the Android deployment pack on Windows, you will need:

- Windows XP/Vista or Windows 7
- LiveCode 4.5.3 or later
- The Java SDK: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- The Android SDK: http://developer.android.com/sdk/index.html

If you are intending to use the Android deployment pack on Mac, you will need:

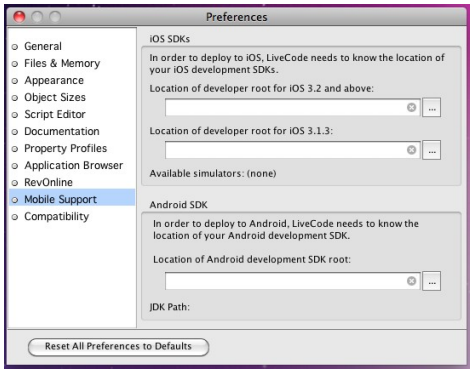
- Mac OS 10.5.x or later
- LiveCode 4.5.3 or later
- The Android SDK: http://developer.android.com/sdk/index.html

After you have installed the pre-requisites make sure you run the Android SDK Manager and have installed the 'SDK Platform Android 2.2, API 8, revision 2' package.

Configuring LiveCode

After you have set up your system with the Java Development Kit and Android Development Kit, it is necessary to inform LiveCode where to find them.

To configure the paths to your installed SDKs, use the *Android* section of the *Mobile Support* pane in *Preferences*.

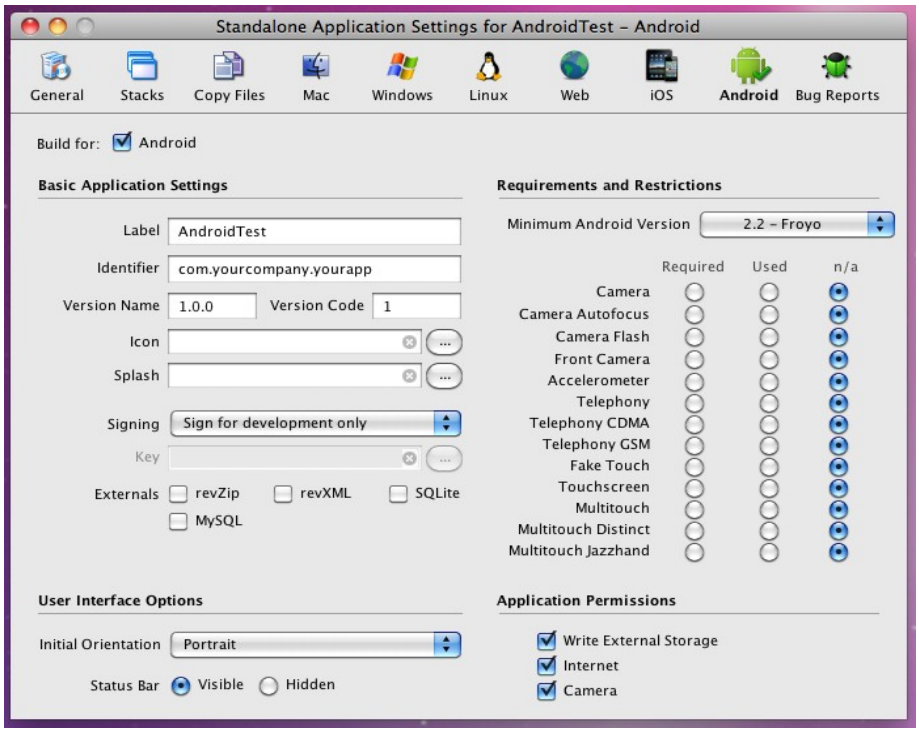


Use this pane to choose the correct SDK paths by clicking the '…' button after the *Android development SDK location* field.

After doing so, the JDK path should be automatically located. If a path fails to appear for the JDK it means you have not correctly installed or configured your JDK.

Configuring an Android standalone

To configure a stack for Android, you use the new Android deployment pane in the *Standalone Application Settings* dialog, available from the *File* menu.



This pane allows you to set the Android-specific options for your application. You can also add files you wish to be included in the bundle using the *Copy Files* pane, and set the name of your application on the *General* pane.

To make a stack build for Android, simply check the *Build for Android* button and configure any options that you wish.

Note: *Making a stack build for Android disables building for any other non-mobile platforms, however this is only true of the standalone's mainstack. If you wish to share code and resources among platforms, simply factor your application into multiple stacks, using a different mainstack for mobile and desktop targets.*

Note: *The* Inclusions, Copy Referenced Files, Bug Reports *and* Stacks *features are* ***not*** *available when building for Android. If you wish to include multiple stackfiles in your application, use the* Copy Files *feature instead.*

Configuring an emulated device

In order to run an Android project, you need either an emulated device running, or a real device configured for debugging connected.

Creating an emulated device is easily done using the Android SDK Manager that you will have previously installed:

- Make sure the SDK Manager is running.
- Choose 'Virtual Devices' from the left-hand list
- Click 'New...'
- Choose a name for your device
- Set the *Target* to at least Android 2.2 – API Level 8
- Fill in an *SD Card* size.
- Enable the *Snapshot* option (this isn't essentially but significantly speeds up subsequent launches of the emulator!)
- Then click *Create AVD*

After you have performed these steps, your newly created device should appear in the list of existing Virtual Devices, from which you can click *Start...* to launch it.

Any running Virtual Devices will appear in the Android Plugin's device list (assuming you have correctly configured the SDK root).

Configuring a real device

Instead of using the emulator, you can also launch LiveCode Android projects on real Android devices after they have been appropriately configured for debugging.

If you are running on Windows then before you can connect to a real device, you need to ensure the appropriate device driver is installed on your development machine. For details of how to do this see here:

http://developer.android.com/sdk/win-usb.html

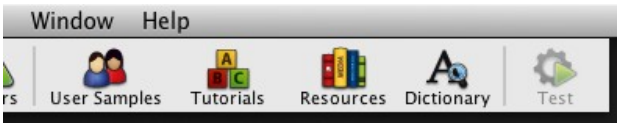
If you are running on Mac, there is no need to install any drivers as it 'just works'.

Once you have any necessary drivers installed, you must then put your device into debugging mode. To do this, go to the home screen, press **MENU**, select **Applications > Development** and enable **USB debugging**.

Finally, simply connect your device via USB to your machine and it should (after a few moments) be available in the Android Plugin's device list (assuming you have correctly configured the SDK root).

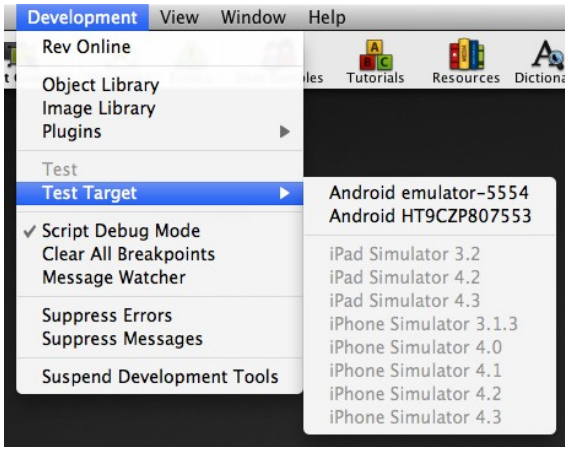
Testing an Android application

Once you have your stack configured for Android, you can test it on either a real Android device or in the Android emulator by using the *Test* button on the menubar:



This button will be enabled for any stack that has been configured for Android deployment, and clicking it will launch the stack on the currently chosen *Test Target*, terminating any running app with the same id as necessary.

You can access the *Test* action from the Development menu. Additionally, this is where you can configure which target Android device to use:



Here you can choose which Android device to use for Android testing. Any setting you choose here will take effect the next time you use the *Test* button or menu-item.

Note: *If the Test button or menu-item remains disabled, even if you have configured a stack for Android deployment, it probably means you haven't configured your SDKs correctly. In this case, check the appropriate settings on the Mobile Support pane of Preferences.*

A First Project

Once you have installed an Android SDK and configured LiveCode for it, it is easy to run a simple project:

1. Create a new main stack via **File > New Mainstack**.
2. Rename your new main stack to *Hello World*
3. Drag and drop a button onto the new main stack, and call it *Click Me*
4. Edit the *Click Me* button script and enter the following:

```
on mouseUp
 answer "Hello World!" with "ok"
end mouseUp
```

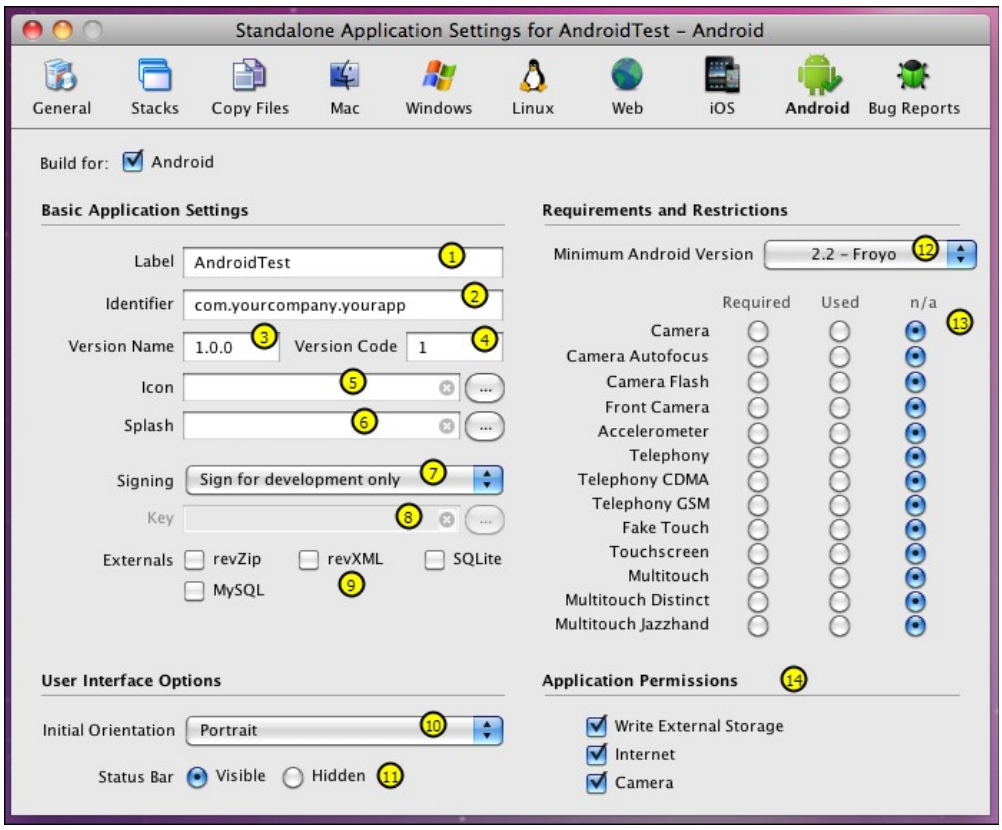
5. Save the *Hello World* stack.
6. Bring up the *Standalone Application Settings* dialog from the *File* menu, switch to the Android pane and make sure 'Build for Android' is checked.
7. Make sure your test stack is active and then click *Test* on the menubar.
8. Click the *Click Me* button in the simulator to see your script in action!

You can try the stack out in different versions of the simulator, simply by selecting the version you want from the *Development* menu.

Configuring an Android Application

Setting manifest options

All Android applications have a manifest that is built into the package which controls many aspects of the applications requirements and functionality. To set the manifest up, you simply use the options presented in the Standalone Builder's Android pane, these will be used to construct a suitable manifest automatically:



Here the numbered items are as follows:

1. The string to display as the label of the application on the Launcher screen.
2. The unique identifier to use for the application, using the standard reverse domain name convention.
3. A human readable version string for the application.
4. An integer indicating the version of the application – this is used by the OS to compare versions of packages as opposed to the human readable string.
5. The PNG image file to use as the icon on the Launcher.
6. The image file to use in the personal and educational splash screens (this is not used when building with a commercial license).
7. Whether APKs are to be signed with the development key, a custom key or no key (this is only used when using 'Save as Standalone application').

8. The key-store file to use to sign the application when “Sign with my key” is selected at step 7 (this is only used when using 'Save as Standalone application').
9. The extensions to include in the application.
    i. Choose 'revZip' if you are using any of the revZip commands and functions.
    ii. Choose 'revXML' if you are using any of the revXML commands and functions.
    iii. Choose 'SQLite' if you are using revDB along with the dbSQLite database driver.
    iv. Choose 'MySQL' if you are using revDB along with the dbMySQLdatabase driver.
10. The initial orientation to start the application up in.
11. The initial visibility of the status bar.
12. The minimum Android version required by the application.
13. The features that should be added to the manifest. A required feature will only be visible to users who have devices that support the feature. A used feature will indicate to the user that this application uses the feature. A used feature will still be visible to devices which do not support the feature.
14. The permissions to be added to the manifest.
    i. Write external storage is required if your app will read or write files on external storage (e.g. an SD card)
    ii. Internet is required if your app is accessing the internet.
    iii. Camera is required if your app is using any camera features.

Adding a launcher icon

All applications currently installed on an Android device are displayed on the launch screen.

To customize the icon used for your application, you should provide an icon image (in PNG format) of size 72x72 and reference it using the appropriate option in the Android standalone settings pane.

Adding a splash image (personal and educational)

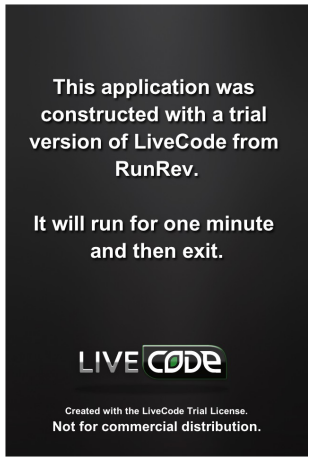
If you are using a personal or educational license, then you are restricted in what can be displayed as the launch image. In this case you should provide a (square) PNG image that will be placed inside a LiveCode branded banner (see below).

We recommend providing an image of 600x600 for the splash – this will give good results when resampled at the various resolutions and sizes required by the different Android devices.

Note: *With these license types, the generated launch image will remain on screen for 5 seconds before being dismissed.*

Adding a default launch image (trial)

If you are evaluating the Android deployment feature using a trial license, then you cannot configure a splash or launch image. Instead, all such applications will be built with the following launch image:



This image will remain on screen for 5 seconds before the application launches, and the application will quit after one minute.

Deployment Features

Standalone builder messages

When building a mobile application for either a device (through *Save as standalone*) or for simulation (by clicking *Simulate)*, messages are sent to the application's main-stack to notify it before building starts, and after build has finished.

Before the application is built the following (optional) message is sent:

savingMobileStandalone *targetType*, *appBundle*

Where *targetType* is either "android release" or "android test", depending on the type of build; and *appBundle* is the path of the application bundle being built.

After the application is built (but before being launched in the simulator), the following (optional) message is sent:

mobileStandaloneSaved *targetType*, *appBundle*

Where the parameters are the same as before except if the build failed, in which case *appBundle* will be empty.

Note that if you make changes to the stack in savingMobileStandalone that you want to appear in the built application, you must save the stack before returning from the handler. The mobile standalone builder uses the stackfile as it is on disk after return from the message to build the app.

General Engine Features

Engine version

The Android engine version is in step with desktop engine version and build number. A substantial subset of the the desktop feature set is available, together with a library of mobile specific functionality.

What doesn't work

The following features have no effect:

- clipboard related syntax and functionality (planned for a future release)
- printing syntax and functionality (planned for a future release)
- setting the mouseLoc (no support on mobile devices)
- socket syntax and functionality (planned for a future release)
- dbPostgreSQL, dbODBC and custom externals (planned for a future release)
- print to pdf (planned for a future release)
- industrial strength encryption and public key cryptography (planned for a future release)
- dbMysql SSL support (planned for a future release)
- paint tools (planned for a future release)
- videoclips/player functionality (planned for a future release)
- revBrowser (substitute mechanism planned for a future release)
- revFont (substitute mechanism planned for a future release)
- drag-drop related syntax and functionality (no support on mobile devices)
- backdrop related syntax and functionality (no support on mobile devices)
- cursor related syntax and functionality (no support on mobile devices)
- revSpeak (no support on mobile devices)
- full screen snapshot commands (planned for a future release)
- visual effects (planned for a future release)
- font querying support (planned for a future release)

What does work

The following things do work as expected:

- rendering of controls with non-system themes (default is Motif theme)
- date and time handling
- gradients, graphic effects and blending
- any non-platform, non-system dependent syntax (maths functions, string processing

functions, behaviors etc.)

- object snapshot commands
- revZip, revXML, dbSqlite and dbMysql

Windowing and Stacks

The Android engine uses a very simple model for window management: only one stack can be displayed at a time.

The stack that is displayed is the most recent one that has been targeted with the **go** command.

The currently active stack will be the target for all mouse and keyboard input, as well as be in receipt of a **resizeStack** message should the orientation or layout of the screen change.

The **modal** command can also still be used, and will cause the calling handler to block until the modal'ed stack is closed as with the normal engine. Note, however, that performing a further **go stack** from a modal'ed stack will cause the new stack to layer above the modal stack – this will likely cause many headaches, so it is probably best to avoid this case!

At this time menus and other related popups will not work correctly, as these are implemented in the engine (essentially) as a specialized form of **go stack** they will cause the current stack to be overlaid completely, with various undesirable side-effects.

Note: *The 'go in window' form of the 'go stack' command will not work correctly in the Android engine and must not be used. Since there is only one stack/window displayed at once on this platform, a generic 'go stack' should be used instead.*

System Dialogs – answer and ask

The Android engine supports a restricted version of the *answer* commands – using the standard Android AlertDialog class.

The answer command can be used in this form:

answer *message* [**with** *button* **and** …] [**titled** *title*]

This will use the Android standard alert popup with the given buttons and title. The last button specified will be marked as the default button.

The ask command can be used in this form:

ask [**question** | **password**] *prompt* **[with** *initialAnswer*] [**titled** *title*]

If neither question nor password is specified, question is assumed. The value entered by the user will be retured in *it*. If the user cancelled the dialog, **the result** will contain *cancel*.

Note: *You cannot nest calls to answer on Android. If you attempt to open an answer dialog while one is showing, the command will return immediately as if the dialog had been cancelled.*

Cross-Mobile Note: *The answer and ask commands work the same way on both iOS and Android.*

Non-file URL access

The Android engine has support for fetching urls, posting to urls and downloading urls in the background. Note that the Android engine does not support libUrl, and as such there are some

differences between url handling compared to the desktop.

The Android engine supports the following non-file URL access methods:

- GET for http, https and ftp URLs
- POST for http and https URLs
- PUT for ftp URLs

Note: *When using URLs for these protocols be aware that the Android system functions used to provide them are much stricter with regards the format of URLs – they must be of the appropriate form as specified by the RFC standards. In particular, in FTP urls, be careful to ensure you urlEncode any username and password fields appropriately (libUrl will allow characters such as '@' in the username portion and still work – Android will not be so forgiving).*

To fetch the google home page you can do:

put url ("http://www.google.com") **into** tGooglePage

To post data to a website, you can use:

post tData **to url** tMyUrl

To upload a file to an FTP server you can use:

put tData **into url** "ftp://ftp.myftpserver.com"

To download a url in the background, you can use:

load url tMyUrl **with message** "myUrlDownloadFinished"

Note that, the callback message received after a **load url** will be of the form:

myUrlDownloadFinished *url*, *status*, *data*

Here, *data* is the actual content of the url that was fetched (assuming an error didn't occur).

Progress updates on ongoing url requests are communicated via the *urlProgress* message. This message is periodically sent to the object whose script initiated the operation. It can have the form:

urlProgress *url,* "contacted"

urlProgress *url,* "requested"

urlProgress *url,* "loading", *bytesReceived* , [*bytesTotal*]

urlProgress *url,* "uploading", *bytesReceived*, [*bytesTotal*]

urlProgress *url,* "downloaded"

urlProgress *url,* "uploaded"

urlProgress *url,* "error", *errorMessage*

Note that pBytesTotal will be empty if the web server does not send the total data size.

You can also download a url direct to a file – this is particularly useful when downloading large files since the normal 'url' chunk downloads into memory. To do this use:

libUrlDownloadToFile *url*, *filename*

Unlike the libUrl command of the same name, this command will block until the download is complete, and will notify progress through the **urlProgress** message as described above.

When using GET and POST with http(s) URLs you can use the *httpHeaders* global property to configure the headers to send. This works the same as the desktop engine, any specified headers overriding those of the same key that would normally be sent, and any new keys being appended.

Cross-Mobile Note: *The url handling functionality works the same way on both iOS and Android.*

Externals

The revZip, revXML, dbSqlite (via revDB) and dbMysql (via revDB) externals can be used on Android.

To include these components, simply check the appropriate boxes on the Android Standalone Settings Pane.

Android Specific Engine Features

This version of the LiveCode Android engine includes a number of features specified to Android devices. These are described in the following sections.

Limitations

Apart from the list of general engine features that do not currently work in the LiveCode Android environment, the current release has the following limitations that we are looking to lift in future releases:

- no ability to configure the status bar visibility
- no access to gps
- no native control support
- no support for clearing pending interactions

Multi-touch events

Touches can be tracked in an application by responding to the following messages:

- **touchStart** *id*
- **touchMove** *id*, *x*, *y*
- **touchEnd** *id*
- **touchRelease** *id*

The *id* parameter is a number which uniquely identifies a sequence of touch messages corresponding to an individual, physical touch action. All such sequences start with a **touchStart** message, have one or more **touchMove** messages and finish with either a **touchEnd** or a **touchRelease** message.

A **touchRelease** message is sent instead of a **touchEnd** message if the touch is cancelled due to an incoming event such as a phone-call.

No two touch sequences will have the same *id*, and it is possible to have multiple (interleaving) such sequences occurring at once. This allows handling of more than one physical touch at once and, for example, allows you to track two fingers moving on the device's screen.

The sequence of touch messages is tied to the control in which the touch started, in much the same way mouse messages are tied to the object a mouse down starts in. The test used to determine what object a touch starts in is identical to that used to determine whether the pointer is inside a control. In particular, invisible and disabled controls will not considered viable candidates.

Cross-Mobile Note: *Touch messages work the same way on both Android and iOS platforms.*

Mouse events

The engine will interpret the first touch sequence in any particular time period as mouse events in the obvious way: the start of a touch corresponding to pressing the primary mouse button, and the end of a touch corresponding to releasing the primary mouse button.

This means that all the standard LiveCode controls will respond in a similar way as they do in the desktop version – in particular, you will receive the standard mouse events and *the mouseLoc* will be kept updated appropriately.

Note that touch messages will still be sent, allowing you to choose how to handle input on a per-control basis.

Cross-Mobile Note: *Mouse messages work the same way on both Android and iOS platforms.*

Motion events

An application can respond to any motion events by using the following messages:

- **motionStart** *motion*
- **motionEnd** *motion*
- **motionRelease** *motion*

Here *motion* is the type of motion detected by the device. At present, the only motion that is generated is “shake”.

When the motion starts, the current card of the defaultStack will receive **motionStart** and when the motion ends it will receive **motionEnd**. In the same vein as the touch events, **motionRelease** is sent instead of **motionEnd** if an event occurs that interrupts the motion (such as a phone call).

Cross-Mobile Note: *Motion messages work the same way on both Android and iOS platforms.*

Hardware button support

When the user presses the hardware 'Back' key, a **backKey** message is sent to the current card of the default stack. If the message is passed or not handle, the engine will automatically quit.

When the user presses the hardware 'Menu' key, a **menuKey** message is sent to the current card of the default stack.

When the user presses the hardware 'Search' key, a **searchKey** message is sent to the current card of the default stack.

Accelerometer support

You can enable or disable the iPhone's internal accelerometer by using:

mobileEnableAccelerometer

mobileDisableAccelerometer

Enabling the accelerometer will cause **accelerationChanged** events to be delivered to the current card of the defaultStack at a frequent interval.

The **accelerationChanged** message takes a single parameter pSample, which consists of four values:

x,y,z,t

Here x, y and z are the acceleration along those axes relative to gravity. The t value is a relative measurement of how much time has passed – you can use the difference between the time values in two **accelerationChanged** events to give an indication of how much time passed between the

samples.

Cross-Mobile Note: *This feature works in the same way on both Android and iOS platforms, although may require different calibrations as the underlying sensors will vary.*

Photo album and camera support

Taking or choosing photos

You can hook into Android's native gallery or camera application by using:

mobilePickPhoto *source*

Here *source* is one of:

- *library* – the user picks a photo using the Android Gallery application
- *album* – the user picks a photo using the Android Gallery application
- *camera* – the user is prompted to take a picture using the Android Camera application

If the source type isn't available on the target device, the command will return with result *"source not available"*. If the user cancels the pick, the command will return with result *"cancel"*. Otherwise a new image object will be created on the current card of the default stack containing the chosen image.

Note: *The image object is cloned from the templateImage, so you can use this to configure settings before calling the picker.*

Keyboard Input

Support for basic (soft) keyboard input is provided automatically. The current Android soft keyboard will be shown when a field is focused, and hidden again when there is no focused field.

Note: *At this time, only simple keyboards that provide a one-to-one mapping between keys and characters will function correctly. General support for Android's rich input method framework is planned for a future release.*

Configuring keyboard type

You can configure the type of keyboard that will be displayed by using the **mobileSetKeyboardType** command:

mobileSetKeyboardType *type*

Where *type* is one of:

- default – the normal keyboard
- alphabet – the alphabetic keyboard
- numeric / decimal – the numeric keyboard with punctuation
- number – the number pad keyboard
- phone – the phone number pad keyboard
- email – the email keyboard

The keyboard type setting takes effect the next time the keyboard is shown – it does not affect the currently displaying keyboard, if any.

If you wish to configure the keyboard options based on the field that is being focused, simply use the commands in an *openField* handler of the given field. The keyboard is only shown after this handler returns, so it is the ideal time to configure it.

Cross-Mobile Note: *You can use the same command to configure the keyboard on both Android and iOS.*

Orientation handling

The Android engine includes support for automatic handling of changes in orientation and in so doing gains use of the smooth Android standard animation rotation animation.

Auto-rotation support

You can configure which orientations your application supports, and also lock and unlock changes in orientation.

The engine will automatically rotate the screen whenever the following are true.

- it detects an orientation change
- the orientation is in the currently configured 'allowed' set
- the orientation lock is off

Such a rotation may result in a *resizeStack* message being sent since rotating at 90 degrees switches width and height.

Querying orientation

You can fetch the current device orientation using the **mobileDeviceOrientation()** function. This returns one of:

- *unknown* – the orientation could not be determined
- *portrait* – the device is being held upward with the home button at the bottom
- *portrait upside down* – the device is being held upward with the home button at the top
- *landscape left* – the device is being held upward with the home button on the left
- *landscape right* – the device is being held upward with the home button on the right
- *face up* – the device is lying flat with the screen upward
- *face down* – the device is lying flat with the screen downward

Similarly, you can fetch the current interface orientation using the **mobileOrientation()** function. This returns one of *portrait, portrait upside down, landscape left* and *landscape right*. With the same meanings as for device orientation.

Controlling auto-rotation

To configure which orientations your application supports use:

mobileSetAllowedOrientations *orientations*

Here *orientations* must be a comma-delimited list consisting of at least one of *portrait*, *portrait upside down*, *landscape left* and *landscape right*. The setting will take effect the next time an orientation change is effected – the interface's orientation will only be changed if the new orientation is among the configured list. You can query the currently allowed orientations with the **mobileAllowedOrientations()** function.

To lock or unlock orientation changes for a time use:

mobileLockOrientation and **mobileUnlockOrientation**

The orientation lock is nestable, and when an unlock request causes the nesting to return to zero, the interface will rotate to match the devices current orientation (assuming it is in the set of allowed orientations). You can query the current orientation lock state with the **mobileOrientationLocked()** function.

Note: *Due to a limitation in the OS, 'landscape left' and 'portrait upside-down' orientations are only supported on Android 2.3 and later.*

Orientation changed notification

An application will receive an ***orientationChanged*** message if the device detects a change in its position relative to the ground, and you can use the **mobileDeviceOrientation()** function to find out the current orientation. This message is sent to the current card of the default stack.

The ***orientationChanged*** message is sent **before** any automatic interface rotation takes place thus changes to the orientation lock state and allowed set can be made at this point and still have an effect. If you wish to perform an action after the interface has been rotated, then either do so on receipt of ***resizeStack***, or by using a *send in 0 millisecs* message.

Cross-Mobile Note: *The orientation feature works in the same way on both Android and iOS platforms.*

Device specific orientations

Internally, the Android engine uses the accelerometer to determine the orientation of a device. However, the accelerometer is not calibrated the same way on every device. For example, consider a standard smart phone that is primarily used in portrait mode. The accelerometer points are typically:

- Portrait – 0
- Reverse landscape – 90
- Reverse portrait – 180
- Landscape 270

Compare this with a standard tablet device which is primarily used in landscape mode:

- Portrait – 90
- Reverse landscape – 180

- Reverse portrait – 270
- Landscape – 0

On certain devices, the orientations do not follow the same order, which causes the orientations to be handled incorrectly. For example, one device tested has the following mapping:

- Portrait – 180
- Reverse landscape – 90
- Reverse portrait – 0
- Landscape – 270

In order to correct orientation handling for this device, you must include a file named lc_device_config.txt in your APK bundle. Do this by adding the file in the Copy Files pane of the Standalone Builder. This file is used to map a device's to orientation to accelerometer value and is parsed by the Android engine on startup. If the current device matches an entry in the file, the orientation settings in that entry will be used.

An entry in the orientation file looks like the following:

> device=<MANUFACTURER>|<MODEL>|<DEVICE>|<VERSION.RELEASE>|<VERSION.INCREMENTAL>
>
> orientation_map=<portrait rotation>,<landscape rotation>,[<portrait reverse orientation>],[<landscape reverse orientation>]

The device entry is used to identify specific devices. The entries are pipe separated and can be fetched using the function **mobileBuildInfo** (see section Device Information). If any of these values are left empty, they will act as a wildcard, matching any device. If all parts of the device line match the current device, then the following lines up to the next device line will be applied.

The orientation map is a comma list of accelerometer values. The first item will be interpreted as the screen rotation when in its default portrait orientation, the second when in its default landscape orientation.

For the aforementioned tablet device, the following line would be used:

> orientation_map=180,270

Without this line, the default handling would assume portrait at 0 degrees and landscape at 270, so reporting landscape orientation correctly, but not portrait. The last two items are optional, and define the accelerometer readings for reverse portrait and reverse landscape.

A default lc_device_config.txt file is included in the runtime folder. The contents of this file are prepended onto any lc_device_config.txt file specified in the Copy Files pane of the Standalone builder, meaning that any user added orientation handling for a specific device will override the defaults.

Resolution handling

The engine makes no attempt to scale or adjust layout of stacks based on the resolution or density of the display the application is running on.

A single pixel on the desktop maps to a single screen pixel on Android, giving you full access to all the pixels on a device's display regardless of what dpi it might have.

When a stack is displayed, it will receive a *resizeStack* message resizing to the full area of the screen (minus status bar).

You can use **the screenRect** and **the working screenRect** properties to find out the current full size of the screen, and the area not including the status bar.

Email composition

Basic support

A version of **revMail** has been implemented that hooks into the iPhone's MessageUI framework. Using this, you can compose a message and request that the user send it using their currently configured mail preferences.

The syntax of **revMail** is:

revMail *toAddress*, [*ccAddress*, [*subject*, [*messageBody*]]]

Where the address fields are comma separated lists of email address. If any of the parameters are not present, the empty string is used instead.

Upon return, **the result** will be set to one of:

- *cancel* – if the user chooses to cancel the send
- *sent* – if the user elected to send the email

Note that once you've called the **revMail** command you have no more control over what the user does with the message – they are free to modify it and the addresses as they see fit.

Advanced support

More complete access to Android's mail composition interface is gained by using one of the following commands:

mobileComposeMail *subject*, [*recipients*, [*ccs*, [*bccs*, [*body*, [*attachments*]]]]]

mobileComposeUnicodeMail *subject*, [*recipients*, [*ccs*, [*bccs*, [*body*, [*attachments*]]]]]

mobileComposeHtmlMail *subject*, [*recipients*, [*ccs*, [*bccs*, [*body*, [*attachments*]]]]]

All commands work the same, except different variants expect varying encodings for the *subject* and *body* parameters:

- *subject* – the subject line of the email. If the Unicode form of the command is used, this should be UTF-16 encoded text.
- *recipients* – a comma -delimited list of email addresses to place in the email's 'To' field.
- *ccs* – a comma-delimited list of email addresses to place in the email's 'CC' field.
- *bccs* – a comma-delimited list of email addresses to place in the email's 'BCC' field.
- *body* – the body text of the email. If the Unicode variant is used this should be UTF-16 encoded text; if the HTML variant is used then this should be HTML.
- *attachments* – either **empty** to send no attachments, a single attachment array or a one-based numeric array of attachment arrays to include.

The attachments parameter consists of either a single array, or an array of arrays listing the attachments to include. A single attachment array should consist of the following keys:

- *data* – the binary data to attach to the email (not needed if *file* present)
- *file* – the filename of the file on disk to attach to the email (not needed if *data* present)
- *type* – the MIME-type of the data.
- *name* – the default name to use for the filename displayed in the email

If you specify a file for the attachment, the engine's does its best to ensure the least amount of memory is used by asking the OS to only load it from disk when needed. Therefore, this should be the preferred method when attaching large amounts of data.

For example, sending a single attachment might be done like this:

```
put "Hello World!" into tAttachment["data"]
put "text/plain" into tAttachment["type"]
put "Greetings.txt" into tAttachment["name"]
iphoneComposeMail tSubject, tTo, tCCs, tBCCs, tBody, tAttachment
```

If multiple attachments are needed, simply build an array of attachment arrays:

```
put "Hello World!" into tAttachments[1]["data"]
put "text/plain" into tAttachments[1]["type"]
put "Greetings.txt" into tAttachments[1]["name"]
put "Goodbye World!" into tAttachments[2]["data"]
put "text/plain" into tAttachments[2]["type"]
put "Farewell.txt" into tAttachments[2]["name"]
mobileComposeMail tSubject, tTo, tCCs, tBCCs, tBody, tAttachments
```

Upon completion of a compose request, **the result** will be set to one of the following:

- *sent* – the email was sent successfully
- *cancel* – the email was not sent, and the user elected not to save it for later

Some devices will not be configured with email sending capability. To determine if the current device is, use the **mobileCanSendMail()** function. This returns true if the mail client is configured.

Cross-Mobile Note: *The compose mail features work in a similar fashion on both Android and iOS.*

File and folder handling

In general the low-level support for handling files and folders in the Android engine is the same as that on the desktop. All the usual syntax associated with such operations will work. Including:

- **open file**/**read**/**write**/**seek**/**close file**
- **delete file**
- **create folder**/**delete folder**
- setting and getting **the folder**

- listing files and folders using **the [detailed] files** and **the [detailed] folders**
- storing and fetching *binfile:* and *file:* urls

However, it is important to be aware that the Android imposes strict controls over what you can and cannot access.

An Android application is installed on the phone in the form of its package (which is essentially a zip file) – in particular, this means that any assets that are included are not available as discrete files directly in the native filesystem. To make this easier to deal with, the engine essentially 'virtualizes' the asset files you include allowing (read-only) manipulation with all the standard LiveCode file and folder handling syntax.

To access the assets you have included within your application, use filenames relative to *specialFolderPath("engine").* For example, to load in a file called 'foo.txt' that you have included in the *Files and Folders* list, use:

put url ("file:" & **specialFolderPath**("engine") & **slash** & "foo.txt") **into** tFileContents

Or if you want to get a list of the image files that you have included within a folder *myimages* in the app package, use something like:

set the folder to specialFolderPath("engine") & **slash** & "myimages"

put the files into tMyImages

Other standard file locations can be queried using the **specialFolderPath()** function. The following paths are supported on Android at this time:

- *engine* – the (virtual) path to the engine and its assets
- *documents* – the path to a folder to use for per-application data files
- *cache* – the path to a folder to use for transient per-application data files

Note: *The Android filesystem is* ***case-sensitive*** *– this is different from (most) Mac installs and Windows so take care to ensure that you consistently use the same casing of filenames when constructing them.*

Cross-Mobile Note: *The special handling the engine does of asset files on Android means that you can use the same code to access such files on both iOS and Android – on both platforms such files are accessible relative to the same base folder* ***specialFolderPath****("engine").*

Basic sound playback support

Basic support for playing sounds has been added using a variant of the **play** command. A single sound can be played at once by using:

play *soundFile* [**looping**]

Executing such a command will first stop any currently playing sound, and then attempt to load the given sound file. If **looping** is specified the sound will repeat forever, or until another sound is played.

If the sound playback could not be started, the command will return "could not play sound" in **the result**.

To stop a sound that is currently playing, simply use:

play empty

The volume at which a sound is played can be controlled via **the playLoudness** global property.

The overall volume of sound playback depends on the current volume setting the user has on their device.

This feature uses the built-in sound playback facilities on Android and as such has support for a variety of formats including MP3's.

You can monitor the current sound being played by using **the sound** global property. This will either return the filename of the sound currently being played, or “done” if there is no sound currently playing.

Cross-Mobile Note: *This feature works in the same way on both Android and iOS platforms, although the list of supported audio formats will vary between devices.*

Video playback support

Basic support for playing videos has been added using a variant of the **play** command. A video file can be played by using:

play (*video-file* | *video-url*)

The video will be played fullscreen, and the command will not return until it is complete, or the user dismisses it.

If a path is specified it will be interpreted as a local file. If a url is specified, then it must be either an 'http', or 'https' url. In this case, the content will be streamed.

The playback uses Android's built-in video playback support and as such can use any video files supported by the device.

Appearance of the controller is tied to **the showController of the templatePlayer**. Changing this property to true or false, will cause the controller to either be shown, or hidden.

When a movie is played without controller, any touch on the screen will result in a ***movieTouched*** message being sent to the object's whose script started the video. The principal purpose of this message is allow the **play stop** command to be used to stop the movie. e.g.

on movieTouched

    play stop

end movieTouched

Note: *The movieTouched message is* ***not*** *sent if the video is played with* ***showController*** *set to true.*

Playing a video can be made to loop by setting **the looping of the templatePlayer** to true before executing the play video command.

URL launching support

Support for launching URLs has been added. The **launch url** command can now be used to request the opening of a given url:

launch url *urlToOpen*

When such a command is executed, the engine first checks to see if an application is available to

handle the URL. If no such application exists, the command returns "no association" in **the result**. If an application is available, the engine requests that it launches with the given url.

Using this syntax it is possible to do things such as:

- open the mobile browser with a given *http:* url
- open the dialer with a given phone number using a *tel:* url

If you are trying to launch a file using an external viewer, you need to make sure the file is visible to the viewer (i.e. not in the apps private file system). A standard approach to this is to store the desired file on the SD card (/sdcard/). If you app is writing to an external location, have the "Write External Storage" option ticked in the standalone builder.

Hardware and system version query support

You can fetch information about the current hardware and system version using the standard LiveCode syntax in the following ways.

To determine what processor an application is running on use **the processor**. For the Android engine this will always be ARM, regardless of whether running on a virtual or real device.

To determine the type of device an application is running on use **the machine**. This will return the (manufacturer's name) for the device. For example, if running on a Google Nexus One, the string will be *Nexus One*.

To determine the version of Android the application is running on, use **the systemVersion**. For example, if the device has Android 2.2 installed, this property will return 2.2; if the device has Android 2.3.1 installed, this property will return 2.3.1.

Querying camera capabilities

To find out the capabilities of the current devices camera(s), use the following function:

mobileCameraFeatures([*camera*])

The *camera* parameter is a string containing either "rear" or "front". In this case, the capabilities of that camera are returned. These take the form of a comma-delimited list of one or more of the following:

- *photo* – the camera is capable of taking photos
- *video* – the camera is capable of recording videos
- *flash* – the camera has a flash that can be turned on or off

If the returned string is empty it means the device does not have that type of camera.

If no camera parameter is specified (or is empty), then a comma-delimited list of one or more of the following is returned:

- *front photo* – the front camera can take photos
- *front video* – the front camera can record video
- *front flash* – the front camera has a flash
- *rear photo* – the rear camera can take photos

- *rear video* – the rear camera can record video
- *rear flash* – the rear camera has a flash

If the returned string is empty it means the device has no cameras.

Note: *At this time, Android can only detect whether there are front and/or back cameras and whether they can take photos.*

Status bar configuration support

You can now configure the status bar that appears at the top of the Android screen.

To control the visibility of the status bar use the following commands:

mobileShowStatusBar

mobileHideStatusBar

These commands cause the status bar to be shown and hidden respectively.

Cross-Mobile Note: *This feature works in the same way on both Android and iOS platforms.*

Device Information

The function **mobileBuildInfo** can be used to fetch information about the current device, such as the manufacturer and device names.

mobileBuildInfo(property)

The property value can be one of the following:

- BOARD - The name of the underlying board, like "goldfish".
- BOOTLOADER - The system bootloader version number.
- BRAND - The brand (e.g., carrier) the software is customized for, if any.
- CPU_ABI - The name of the instruction set (CPU type + ABI convention) of native code.
- CPU_ABI2 - The name of the second instruction set (CPU type + ABI convention) of native code.
- DEVICE - The name of the industrial design.
- DISPLAY - A build ID string meant for displaying to the user.
- FINGERPRINT - A string that uniquely identifies this build.
- HARDWARE - The name of the hardware (from the kernel command line or /proc).
- HOST
- ID - Either a change list number, or a label like "M4-rc20".
- MANUFATURER - The manufacturer of the product/hardware.
- MODEL - The end-user-visible name for the end product.
- PRODUCT - The name of the overall product.
- RADIO - The radio firmware version number.

- SERIAL - A hardware serial number, if available.
- TAGS - Comma-separated tags describing the build, like "unsigned,debug".
- TIME
- TYPE - The type of build, like "user" or "eng".
- USER

In App Purchasing

Syntax

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

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

Commands & Functions

To determine if in app purchasing is available use:

mobileCanMakePurchase()

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

Throughout the purchase process, the AppStore 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:

mobileEnablePurchaseUpdates

mobileDisablePurchaseUpdates

To create a new purchase use:

mobilePurchaseCreate *productID*

The *productID* is the identifier of the in-app purchase you created and wish to purchase. A purchaseID is placed in the result which is used to identify the purchase.

To query the status of an active purchase use:

mobilePurchaseState(*purchaseID*)

The *purchaseID* is the identifier of the purchase request. One of the following is returned

- *initialized* - the purchase request has been created but not sent. In this state additional properties such as the item quantity can be set.

- *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 **mobilePurchaseConfirmDelivery** command.
- *complete* - the purchase has now been paid for and delivered
- *restored* - the purchase has been restored after a call to **mobileRestorePurchases**. The purchase should now be delivered to the user and confirmed via the **mobilePurchaseConfirmDelivery** command.
- *cancelled* - the purchase was cancelled by the user before payment was received
- *refunded* - the payment for the item was refunded to the user
- *unverified* - the payment request message could not be verified (the public key was not available, or the verification check failed). Call **mobilePurchaseVerify** to clear this status.
- *error* - An error occurred during the payment request. More detailed information is available from the **mobilePurchaseError** function

To get a list of all known active purchases use:

mobilePurchases()

It returns a return-separated list of purchase identifiers, of restored or newly bought purchases which have yet to be confirmed as complete.

Before sending an your purchase request using the **mobilePurchaseSendRequest,** you can configure aspects of it by setting certain properties. This is done using:

mobilePurchaseSet *purchaseID*, *property*, *value*

The parameters are as follows:

- *purchaseId* - the identifier of the purchase request to be modified
- *property* - the name of the property to be set
- *value* - the value to set the property to

Properties which can be set include:

- *dev*eloperPayload - 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. Defaults to empty.

As well as setting properties, you can also retrieve them using:

mobilePurchaseGet(*purchaseID, property)*

The parameters are as follows:

- *purchaseID* - the identifier of the purchase request
- *property* - the name of the purchase request property to get

Properties which can be queried include:

- *productID* - identifier of the purchased product
- *purchaseDate* - date the purchase / restore request was sent
- *transactionIdentifier* - the unique identifier for a successful purchase / restore

- *developerPayload* - the developer payload value that was sent with the original purchase request
- *signedData* - a string containing detailed information about the purchase request response, in JSON format. This is signed by the Android Market using the private key application developer's publisher account, the public half of the key pair can then be used to verify that the message came from the Android Market.
- *signature* - the cryptographic signature of the signedData, in base64 encoding

Once you have created and configured your purchase you can send it to the AppStore to start the purchase using:

mobilePurchaseSendRequest *purchaseID*

Here, *purchaseID* is the identifier of the purchase request. This command should only be called on a purchase request in the 'initialized' state.

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 AppStore once you have completely fulfilled the purchase using:

mobilePurchaseConfirmDelivery *purchaseID*

Here, *purchaseID* is the identifier of the purchase request.

mobilePurchaseConfirmDelivery should only be called on a purchase request in the 'paymentReceived' or 'restored' state. 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 **mobileEnableUpdates** command.

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

mobileRestorePurchases

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:

mobilePurchaseError(*purchaseID*)

The *purchaseID* is the identifier of the purchase request. It returns the error information for purchase requests in the "error" state.

If you wish to confirm or reject a purchase in the unverified state use:

mobilePurchaseVerify *purchaseId, verified*

Here, the parameters are:

- *purchaseId* - the identifier of the purchase request
- *verified* - boolean: if true, a new purchase update will be sent with the status updated to show the state of the purchase request if false, a new purchase update will be sent with the purchase in the error state

Messages

The AppStore sends **purchaseStateUpdate** messages to notifies your app of any changes in state to

the purchase request. These messages continue until you notify the AppStore that the purchase is complete or it is cancelled.

purchaseStateUpdate *purchaseID*, *state*

The *state* can be any one of the following:

- *initialized* - the purchase request has been created but not sent. In this state additional properties such as the item quantity can be set.
- *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 **mobilePurchaseConfirmDelivery** command
- *complete* - the purchase has now been paid for and delivered
- *restored* - the purchase has been restored after a call to **mobileRestorePurchases**. The purchase should now be delivered to the user and confirmed via the **mobilePurchaseConfirmDelivery** 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 **mobilePurchaseError** function

Noteworthy Changes

OpenGL Compositor (5.0.1-dp-1)

You can now use the 'opengl' compositor type on Android. This feature is still in development. There are the following known issues:

1. Visual effects will not work when compositorType is "opengl".
2. When switching between opengl and non-opengl the screen will render black briefly.
3. The engine will output various information related to OpenGL usage to the device log - in particular related to supported 'EGL configurations'. This information will be useful in helping diagnose problems if OpenGL mode does not work on a specific device.
4. OpenGL mode has not been fully tested with all Android features

Change Logs and History

Engine Change History

pre-release-1 (2011-02-21)	MW	Initial version.
pre-release-2 (2011-02-23)	MW	Fixed bug – initial touch ignored Fixed bug – multiple touches not reported correctly
pre-release-3 (2011-03-08)	MW	Added support for unique identifiers for each package allowing multiple LiveCode Android apps on a device. Added support for asset file inclusion, allowing files to be bundles with a LiveCode app. Added support for 'ask' and 'ask password' dialogs Added support for the accelerometer (if present) Added support for basic sound playback
4.6.1-rc-1 (2011-04-19)	MW	Added support for unicode text rendering Added support for non-file URL access Added support for picking photos from library and camera Added support for orientation changes Added support for mail composition Added support for launch url Added support revXML, revZip, dbSQLite and dbMySQL Added support for 'backKey' message when back button pressed Added support for cache and documents to specialFolderPath Fixed bug with startup / shutdown when returning from home Fixed bug with quit command Fixed numerous graphic rendering issues
4.6.1-gm-1 (2011-04-25)	MW	Added support for basic keyboard input. Fixed bug with the mouseLoc being vertically displaced. Fixed bug with mouseRelease not being sent when touch cancelled.
4.6.1-gm-2 (2011-05-04)	MW	Fixed bug with keyboard appearing even if no focusable controls. Fixed bug causing app to crash on exit if global variables used (9526)
4.6.2-dp-1 (2011-06-01)	MW	Fixed a number of intermittant crashes and stability issues. Fixed bug with camera not working (9522) Fixed bug with orientation on tablets (9540)
4.6.2-rc-1 (2011-06-08)	MW	Fixed bug with 'folder' needing trailing slash for engine path (9565)
4.6.2-rc-2 (2011-06-15)	MW	Ask and answer dialogs now respond to the back button (cause cancellation). Added support for mobileShowStatusbar / mobileHideStatusbar. Added support for mobileCanSendMail. Fixed bug with orientation reporting on landscape devices.
4.6.2-gm-1 (2011-06-20)	MW	Fixed bug with reversal of red/blue in bitmap effects.
4.6.3-dp-1 (2011-07-01)	MM	Fixed bug with mobilePickPhoto crashing (9522). Fixed bug with non-alphabetic keyboard types (9594).
4.6.3-dp-2 (2011-07-11)	MM	No changes.

4.6.3-dp-3(2011-07-13)	MM	No changes.
4.6.3-rc-1(2011-07-15)	MM	Added support for streaming of hosted videos. (Bug 9614). Fixed bug with certain graphics rendering incorrectly (9623). Addressed bug with incorrect orientation handling on certain devices (9540). Added function mobileDeviceInfo.
4.6.3-gm-1(2011-07-19)	MM	No changes.
4.6.3-gm-2(2011-07-26)	MM	Fixed bug with opening SQLite databases crashing (9630). Fixed bug with URL downloads sporadically corrupting.
4.6.4-dp-1 (2011-08-10)	MM	No changes.
4.6.4-dp-2 (2011-08-16)	MM	Added support for the mouse function.
4.6.4-dp-3 (2011-08-22)	MM	No changes.
4.6.4-rc-1 (2011-08-26)	MM	Added support for the networkInterfaces property.
4.6.4-rc-2 (2011-09-02)	MM	Fixed bug with color's swapping in patterns (9702).
4.6.4-gm-1 (2011-09-06)	MM	No changes.
4.6.4-gm-2 (2011-09-09)	MM	Updated launch URL to handle a wider range of file types (Bug 9713).
4.6.4-gm-3 (2011-09-15)	MM	No changes.
5.0.0-dp-1 (2011-09-15)	MM	Support added for the standard desktop visual effects. Graphics architecture modified to support software accelerated rendering. See main release note for full details.
5.0.0-dp-2 (2011-09-22)	MM	No changes.
5.0.0-dp-3 (2011-09-27)	MM	No changes.
5.0.0-dp-4 (2011-10-03)	MM	Fixed bug - card/stack initialized with wrong size during Android startup. Fixed bug- incorrect region used when updating screen on Android (9772)
5.0.0-rc-4 (2011-10-06)	MM	Fixed bug - low memory warning crashes Android. Fixed bug - import/export snapshot inverts colors on Android (9779). Fixed bug - bitmap effects with non-copy blendMode don't render correctly on Android (9771).
5.0.0-rc-2 (2011-10-08)	MM	No changes.
5.0.0-gm-1 (2011-10-10)	MM	No changes.
5.0.0-dp-1 (2011-10-19)	MM	Implemented OpenGL compositor. Fixed bug - memory leak when redrawing on Android devices in certain cases. Fixed bug - visual effects between stacks do not ensure target stack is resized correctly before playing on mobile.
5.0.1-dp-2 (2011-10-26)	MM	Implemented in-app purchasing.
5.0.1-dp-3 (2011-11-07)	MM	Fixed bug – opaque graphics render incorrectly using OpenGL compositor (9837).
5.0.1-rc-1 (2011-11-16)	MM	Fixed bug - black screen when switching between OpenGL and Bitmap modes on Android. Fixed bug - visual effects don't work in OpenGL mode on Android.
5.0.1-rc-2 (2011-11-21)	MM	Added support for the menu and search hardware buttons.
5.0.1-gm-1 (2011-11-23)	MM	No changes.

Deployment Change History

pre-release-1 (2011-02-21)	MW	Initial version.
pre-release-2 (2011-02-23)	MW	No changes.
pre-release-3 (2011-03-08)	MW	Added support for unique package names Added support for assets
4.6.1-rc-1 (2011-04-19)	MW	Integrated deployment into the IDE.
4.6.1-gm-1 (2011-04-25)	MW	No changes.
4.6.1-gm-2 (2011-05-04)	MW	Fixed bug with building standalones when not using a commercial license.
4.6.2-dp-1 (2011-06-01)	MW	Fixed bug where APKs would be signed with both a debug and distribution key. Fixed bug with detection of appropriate JDK on 64-bit Windows.
4.6.2-rc-1 (2011-06-08)	MW	Fixed bug with icon not being included in APK if not absolute path (9568)
4.6.2-rc-2 (2011-06-15)	MW	No changes.
4.6.2-gm-1 (2011-06-20)	MW	Fixed bug with failing to report an error when signing an APK if the name or password are incorrect.
4.6.3-dp-1 (2011-07-01)	MM	Updated standalone builder to allow setting of manifest permissions and features. Updated standalone builder to allow the signing of APKs with debug key, distribution key or no key. Fixed bug in standalone builder with APK signing errors not being reported.
4.6.3-dp-2 (2011-07-11)	MM	No changes.
4.6.3-dp-3 (2011-07-13)	MM	No changes.
4.6.3-rc-1(2011-07-15)	MM	No changes.
4.6.3-gm-1(2011-07-19)	MM	A default lc_device_config.txt is prepended onto user lc_device_config.txt file (if applicable).
4.6.3-gm-2(2011-07-26)	MM	Fixed bug with Standalone Builder not finding keystore files when relative to the current stack (9633).
4.6.4-dp-1 (2011-08-10)	MM	Updated the standalone build process to handle the latest Android SDK directory structure.
4.6.4-dp-2 (2011-08-16)	MM	Updated standalone builder to allow building with JDK 1.7. Fixed bug with splash screen inclusion for personal and educational licenses.
4.6.4-dp-3 (2011-08-22)	MM	No changes.
4.6.4-rc-1 (2011-08-26)	MM	No changes.
4.6.4-rc-2 (2011-09-02)	MM	No changes.
4.6.4-gm-1 (2011-09-06)	MM	No changes.
4.6.4-gm-2 (2011-09-09)	MM	No changes.
4.6.4-gm-3 (2011-09-15)	MM	No changes.
5.0.0-dp-1 (2011-09-15)	MM	No changes.
5.0.0-dp-2 (2011-09-22)	MM	No changes.
5.0.0-dp-3 (2011-09-27)	MM	No changes.
5.0.0-dp-4 (2011-10-03)	MM	No changes.
5.0.0-rc-1 (2011-10-06)	MM	No changes.
5.0.0-rc-2 (2011-10-08)	MM	No changes.
5.0.0-gm-1 (2011-10-10)	MM	No changes.

5.0.1-dp-1 (2011-10-19)	MM	No changes.
5.0.1-dp-2 (2011-10-26)	MM	Updated the standalone builder to allow specifying of the public key to verify in-app purchases against.
5.0.1-dp-3 (2011-11-07)	MM	Updated standalone builder to allow builds to be stored on the SD card as well as device.
5.0.1-rc-1 (2011-11-16)	MM	No changes.
5.0.1-rc-2 (2011-11-21)	MM	No changes.
5.0.1-gm-1 (2011-11-23)	MM	No changes.

Document Change History

Revision 1 (2011-02-21)	MW	Initial version.
Revision 2 (2011-02-23)	MW	No changes.
Revision 3 (2011-03-08)	MW	Added section on accelerometer support Added section on basic sound playback support Updated section on file and folder handling to mention accessing assets Updated section dialogs to include ask
Revision 4 (2011-04-19)	MW	Completely revised.
Revision 5 (2011-04-25)	MW	Added section on keyboard support.
Revision 6 (2011-05-04)	MW	No changes.
Revision 7 (2011-06-01)	MW	No changes.
Revision 8 (2011-06-08)	MW	No changes.
Revision 9 (2011-06-15)	MW	Added section on status bar configuration support. Updated section on mail handling to mention CanSendMail.
Revision 10 (2011-06-20)	MW	No changes.
Revision 11 (2011-07-01)	MM	Updated section on setting manifest options. Updated section on configuring Android standalone. Added section on video playback support.
Revision 12 (2011-07-11)	MM	Removed section on playing HTTP videos.
Revision 13 (2011-07-13)	MM	No changes.
Revision 14 (2011-07-15)	MM	Added section on playing HTTP videos. Added section Device specific orientation. Added section Device information.
Revision 15 (2011-07-19)	MM	Updated section Device specific orientation.
Revision 16 (2011-07-26)	MM	No changes.
Revision 17 (2011-08-10)	MM	No changes.
Revision 18 (2011-08-16)	MM	No changes.
Revision 19 (2011-08-22)	MM	No changes.
Revision 20 (2011-08-26)	MM	Updated section Device Information.
Revision 21 (2011-09-02)	MM	No changes.
Revision 22 (2011-09-06)	MM	No changes.
Revision 23 (2011-09-09)	MM	Updated section section “URL launching support”.
Revision 24 (2011-09-15)	MM	No changes.
Revision 25 (2011-09-15)	MM	No changes.
Revision 26 (2011-09-22)	MM	No changes.

Revision 27 (2011-09-27)	MM	No changes.
Revision 28 (2011-10-03)	MM	No changes.
Revision 29 (2011-10-06)	MM	No changes.
Revision 30 (2011-10-08)	MM	No changes.
Revision 31 (2011-10-10)	MM	No changes.
Revision 32 (2011-10-19)	MM	Added section “OpenGL Compositor”.
Revision 33 (2011-10-26)	MM	Added section “In-app purchasing”
Revision 34 (2011-11-07)	MM	No changes.
Revision 35 (2011-11-16)	MM	No changes.
Revision 36 (2011-11-20)	MM	Renamed section “Hardware back key” support to “Hardware button support”. Added details of “menuKey” and “searchKey” messages to section “Hardware button support”.
Revision 37 (2011-11-23)	MM	No changes.