Android Deployment Release Notes

Table of Contents

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.*

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

http://www.runrev.com/links/setup-android- *(Under development)*

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

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

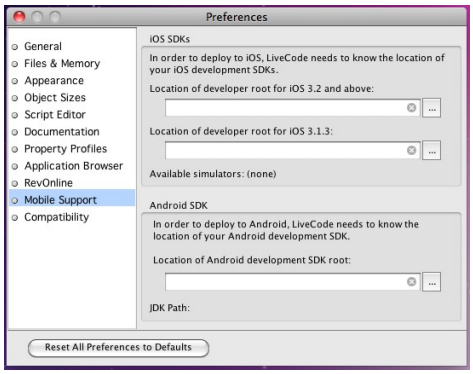
- LiveCode 5.5.2 or later
- The Java SDK: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- The Android SDK: http://developer.android.com/sdk/index.html

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*.



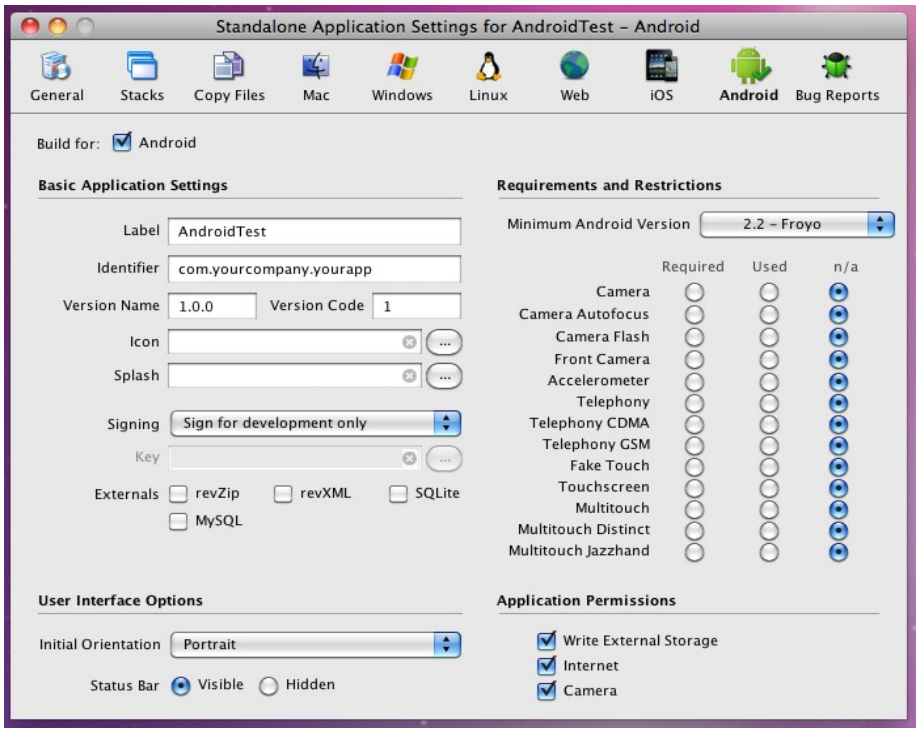
Use this pane to choose the correct SDK paths by clicking the '…' button after the *Android development SDK location* field.

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.

If you are developing on Linux you may have to manually set the path of the 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'.

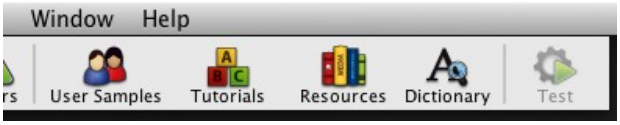
Once you have any necessary drivers installed, you must then put your device into debugging mode. To do this, go to the home screen, press **MENU**, select **Applications > Development** and enable **USB debugging**.

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).

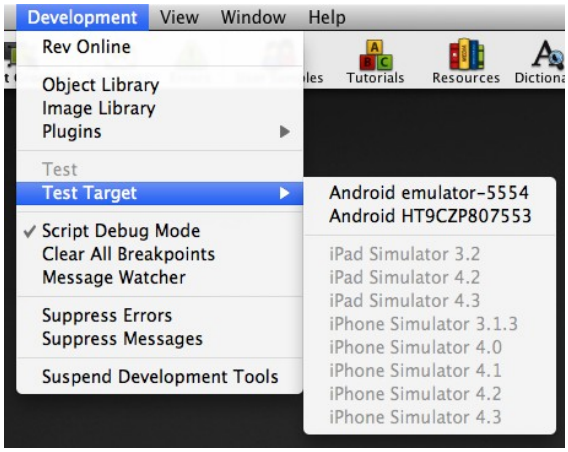
Testing an Android application

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
```

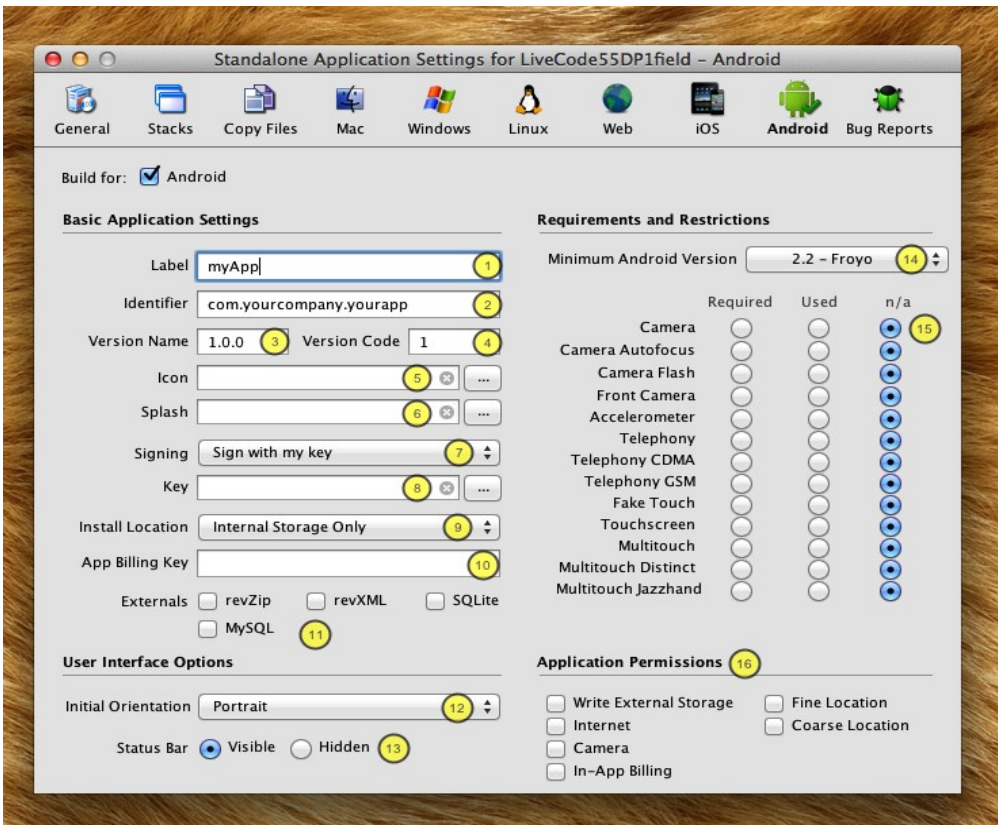
5. Save the *Hello World* stack.
6. Bring up the *Standalone Application Settings* dialog from the *File* menu, switch to the Android pane and make sure 'Build for Android' is checked.
7. Make sure your test stack is active and then click *Test* on the menubar.
8. Click the *Click Me* button in the simulator 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 preference for application storage on the device – This can be *internal storage only* (device memory), allow external storage (user can select to install to device memory or SD card), *prefer external storage* (install to SD card)
10. The Android Marketplace billing key used by the In-App purchasing mechanism. If specified, verification of purchase is automatic. Verification will have to be done manually in script if left blank.
11. 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.
12. The initial orientation to start the application up in.
13. The initial visibility of the status bar.
14. The minimum Android version required by the application.
15. 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.
16. 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.
    iv. In-App Billing is required if you wish to use In-App purchasing
    v. Fine Location is required if you wish to use GPS to triangulate device location (requires course location)
    vi. Coarse location is required if you wish to use mobile networks to triangulate device location

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

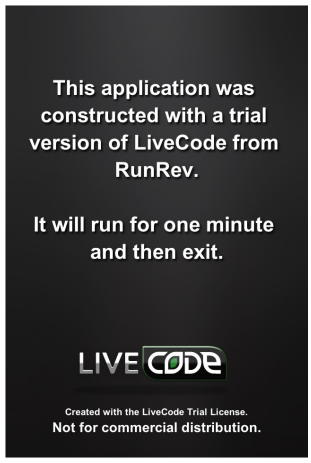
as the launch image. In this case you should provide a (square) PNG image that will be placed inside a LiveCode branded banner (see below).

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.

Adding custom fonts

In LiveCode 5.5 the ability was introduced to allow applications to bundle custom fonts which then become available to the app (and only that app) while it is running.

To take advantage of this feature, all you need to do is reference the files of any fonts you wish to include in the *Copy Files* pane. These files can either be a direct file reference, or contained in one

of the folder references. The Standalone Builder will treat any files that end with the extension *ttf* or *ttc* as font files to use in this way.

Any fonts included in this way will appear in ***the fontNames*** and can be used in the same way as any other font on the system.

Important: *Make sure you have an appropriate license for the fonts you choose to bundle with your app like you would any other media such as sounds, images and videos.*

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 (use native browser control instead)
- revFont (use 'custom font inclusion' mechanism instead)
- 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)

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* | **with hint** *hint*] [**titled** *title*]

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

The *hint* can be used to specify background text that will disappear as soon as the user enters data. The *hint* will never be returned.

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

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 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.

System alert support

To perform a system alert, use the **beep** command.

Vibration support

mobileVibrate *[numberOfTimes]*

To make the device vibrate, use the command **mobileVibrate.** The parameter *numberOfTimes* determines the number of times you wish the device to vibrate. This defaults to 1.

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

Accelerometer support

Note that as of 5.5-dp-1, this syntax has been deprecated in favour of the new generic sensor syntax. See the section “Sensor tracking”.

You can enable or disable the device'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 four parameters ***(Note this has changed from a single parameter in 5.5-dp-1):***

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.*

Location tracking (GPS)

Note that as of 5.5-dp-1, this syntax has been deprecated in favour of the new generic sensor syntax. See the section “Sensor tracking”.

Determining support

To determine if a device has the necessary hardware support for tracking location using GPS use the **mobileCanTrackLocation()** function.

This returns *true* if location can be tracked, or *false* otherwise.

Activating and deactivating tracking

Assuming the hardware is present, tracking of the current location of the device can be activated and deactivated by using:

mobileStartTrackingLocation

mobileStopTrackingLocation

Starting to track location may request permission from the user to access the GPS hardware depending on system settings.

Detection location changes

You can detect changes in location by handling the ***locationChanged*** message. This message is sent to the current card of the default stack.

If location tracking cannot be started (typically due to the user 'not allowing' access to CoreLocation) then a ***trackingError*** message is sent instead.

Querying the location

While location tracking is active, the current location of the device can be fetched by using the **mobileCurrentLocation**() function.

If location tracking has not been enabled this function returns empty.

If location tracking is active then it returns an array with the following keys:

- *horizontal accuracy* – the maximum error in meters of the position indicated by *longitude* and *latitude*
- *latitude* – the latitude of the current location, measured in degrees relative to the equator. Positive values indicate positions in the Northern Hemisphere, negative values in the Southern.
- *longitude* – the longitude of the current location, measured in degrees relative to the zero meridian. Positive values extend east of the meridian, negative values extend west.
- *vertical accuracy* – the maximum error in meters of the *altitude* value.
- *altitude* – the distance in meters of the height of the device relative to sea-level. Positive values extend upward of sea-level, negative values downward.
- *timestamp* – the time at which the measurement was taken, in seconds since 1970.

If the latitude and longitude could not be measured, those keys together with the *horizontal accuracy* key will not be present. If the altitude could not be measured, that key together with the *vertical* accuracy will not be present.

Heading tracking (digital compass)

Note that as of 5.5-dp-1, this syntax has been deprecated in favour of the new generic sensor syntax. See the section “Sensor tracking”.

Determining support

To determine if a device has the necessary hardware support for tracking heading using a digital compass use the **mobileCanTrackHeading()** function.

This returns *true* if location can be tracked, or *false* otherwise.

Activating and deactivating tracking

Assuming the hardware is present, tracking of the current heading of the device can be activated and deactivated by using:

mobileStartTrackingHeading

mobileStopTrackingHeading

Starting to track heading may request the user to calibrate the magnetometer, see the calibration section for more details.

Detection heading changes

You can detect changes in heading by handling the ***headingChanged*** message. This message is sent to the current card of the default stack.

If heading tracking cannot be started (typically due to a lack of calibration) then a ***trackingError*** message is sent instead.

Querying the heading

While heading tracking is active, the current heading of the device can be fetched by using the **mobileCurrentHeading**() function.

If heading tracking has not been enabled this function returns empty.

If heading tracking is active then it returns an array with the following keys:

- *accuracy* - The maximum deviation (measured in degrees) between the reported heading and true geomagnetic heading. The lower the value, the more accurate the reading.
- *magnetic heading* - The heading (measured in degrees) relative to magnetic north.
- *true heading* - The heading (measured in degrees) relative to true north. If the true heading could not be calculated (usually due to heading tracking not being enabled, or lack of calibration), this key will not be present.
- *heading* - The true heading if available, otherwise the magnetic heading.
- *x, y, z* - The geomagnetic data (measured in microteslas) for each of the x, y and z axes.
- *timestamp - The time at which the measurement was taken, in seconds since 1970.*

Sensor tracking

As of LiveCode 5.5-dp-1, sensor support has been unified into a new set of easy to use syntax. Four different sensor can be tracked:

- **location –** tracks the location of the device using either GPS or network triangulation
- **heading –** tracks the heading of the device using the digital compass
- **acceleration –** tracks the devices motion using the accelerometer
- **rotation rate –** tracks the rotation of the device

The names detailed in bold will be used to reference the sensors.

Sensor availability

mobileSensorAvailable(*sensor*)

The function **mobileSensorAvailable** will return true or false depending upon the availability of the given sensor. Here, *sensor* is the name of the sensor you wish to check as detailed in the previous section.

Start tracking sensor

mobileStartTrackingSensor *sensor, [loosely]*

If a sensor is available, you can start tracking it using the command **mobileStartTrackingSensor.** Once tracking a sensor, periodic messages will be sent to the card specifying any changes. This also enables you to query the reading of a sensor at any point.

The parameter *loosely i*s a boolean determining how detailed the readings from the sensors should be.

- true - readings will be determined without using accurate (but power consuming) sources such as GPS
- false - readings will be determined using accurate(but power consuming) sources such as GPS

Stop tracking sensor

mobileStopTrackingSensor *sensor*

You can stop tracking a sensor at any point using the command **mobileStartTrackingSensor.** This ill mean that the periodic update messages will no longer be dent and that you can no longer query the sensor for readings.

Sensor update messages

Once **mobileStartTrackingSensor** has been called, update messages will be sent to the current card, detailing the sensors latest reading.

locationChanged *latitude, longitude, altitude*

- *latitude* – the latitude of the device
- *longitude* – the longitude of the device
- *altitude* – the altitude of the device

headingChanged *heading*

- *heading* - the heading of the device, in degrees relative to true north if available, otherwise relative to magnetic north

accelerationChanged *x, y, z*

- *x* - the rate of acceleration around the x axis, in radians/second
- *y* - the rate of acceleration around the x axis, in radians/second
- *z* - the rate of acceleration around the x axis, in radians/second

rotationRateChanged *x, y, z*

- *x* - the rate of rotation around the x axis, in radians/second
- *y* - the rate of rotation around the y axis, in radians/second
- *z* - the rate of rotation around the z axis, in radians/second

If at any point there is an error tracking one of the sensors, the **trackingError** message will be sent.

trackingError *sensor*, *errorMessage*

Getting a sensor reading

In addition to the update messages that are sent, you can get the reading of any sensor you are tracking using the function **mobileGetSensorReading**

mobileSensorReading(*sensor, [detailed]*)

The boolean parameter *detailed* determines the amount of detail present in the data returned. If this is false, the data returned is a comma separated list. If true, an array is returned. By default, *detailed* is false.

The data returned depends upon the sensor.

Location - a comma separated list of the latitude, longitude and altitude of the device. If *detailed* is true an array containing the keys latitude, longitude, altitude, time stamp, horizontal accuracy, vertical accuracy, speed and course is returned. If the latitude and longitude could not be measured, those values together with the *horizontal accuracy* key will not be present. If the altitude could not be measured, that value together with the *vertical* accuracy will not be present.

Heading - the heading of the device in degrees. If *detailed* is true an array containing the keys heading, magnetic heading, true heading, time stamp, x, y, z and accuracy is returned.

Acceleration - a comma separated list of the acceleration in the x, y and z axes. If *detailed* is true an array containing the keys x, y, z and timestamp is returned.

Rotation Rate - a comma separated list of the rate of rotation around the x, y and z axes. *If detailed* is true an array containing the keys x, y, z and timestamp is returned.

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.*

Saving photos to the users album

You can save an image to the user's photo album by using:

mobileExportImageToAlbum *imageTextOrControl, [filename]*

Where *imageTextOrControl* is one of:

- the binary data of an image (the 'text') in PNG, GIF or JPEG format
- a long id of an image object containing an image in PNG, GIF or JPEG format

The optional *filename* is the name under which the image file is to be saved on the Android file system. A random name is generated if a name is not specified.

The command will return empty in **the result** if exporting succeeded. Otherwise it will return one of:

- *could not find image* – the image object could not be found
- *not an image* – the object was not an image
- *not a supported format* – the image object in not of PNG, GIF or JPEG format
- *export failed* – an error occurred while trying to save the image to the album

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.*

Activation notifications

The following messages will be sent to the current card of the default stack when the keyboard is shown or hidden:

keyboardActivated

keyboardDeactivated

Handle these messages to move controls or change the display layout to take account of the restricted screen area that will be available.

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, the working screenRect** and **the effective working screenRect** properties to find out the current full size of the screen, the area not including the status bar and the area not including the status bar and keyboard respectively.

The pixel density of the devices screen can be queried using **mobilePixelDensity()**. This function returns the number of hardware pixels per logical pixels.

Text messaging support

Use the command **mobileComposeTextMessage** to launch the default text messaging app.

mobileComposeTextMessage *recipients, [body]*

The *recipients* is a comma separated list of phone numbers you want the message to be sent to. The optional *body* is the content of the message you wish to sent.

Note that once you've called the **mobileComposeTextMessage** 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.

Upon completion of a compose request, the result is set to one of the following:

- sent - the text was sent successfully
- cancel - the text was not sent, and the user elected not to save it for later
- failed - the text could not be sent
- false - the device does not have text messaging functionality

You can determine if the device has the text messaging client configured using the function **mobileCanComposeTextMessage()**. This returns true if the client is configured.

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.

As it is not possible to determine if the user sent the email or not, upon return, **the result** will be *unknown*.

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

As it is not possible to determine if the user sent the email or not, upon return, **the result** will be *unknown*.

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.*

Multi-channel sound support

In addition to basic sound playback support, there is also support for playing sounds on different channels.

Playing Sounds

To play a sound on a given channel use the following command:

mobilePlaySoundOnChannel *sound*, *channel*, *type*

Where *sound* is the sound file you wish to play, *channel* is the name of the channel to play it on and

type is one of:

- *now* – play the sound immediately, replacing any current sound (and queued sound) on the channel.
- *next* – queue the sound to play immediately after the current sound, replacing any previously queued sound. If no sound is playing the sound is prepared to play now, but the channel is immediately paused – this case allows a sound to be prepared in advance of it being needed.
- *looping* – play the sound immediately, replacing any current sound (and queued sound) on the channel, and make it loop indefinitely.

If a sound channel with the given name doesn't exist, a new one is created. When queuing a sound using *next*, the engine will 'pre-prepare' the sound long before the current sound is played, this ensures minimal latency between the current sound ending and the next one beginning.

If an empty string is passed as the sound parameter, the current and scheduled sound on the given channel will be stopped and cleared.

When a sound has finished playing naturally (not stopped/replaced) on a given channel, a ***soundFinishedOnChannel*** message is sent to the object which played the sound:

soundFinishedOnChannel *channel*, *sound*

The message is sent after the switch has occurred between a current and next sound on the given channel. This makes it is an ideal opportunity to schedule the next sound on the channel, thus allowing continuous and seamless playback of sounds.

To stop the currently playing sound, and to clear any scheduled sound, on a given channel use:

mobileStopPlayingOnChannel *channel*

To pause the currently playing sound on a given channel use:

mobilePausePlayingOnChannel *channel*

To resume the current sound's playback on a given channel use:

mobileResumePlayingOnChannel *channel*

Channel Properties

To control the volume of a given sound channel use the following:

mobileSetSoundChannelVolume *channel*, *volume*

mobileSoundChannelVolume(*channel***)**

Here *channel* is the channel to affect, and *volume* is an integer between 0 and 100 where 0 is no volume, 100 is full volume.

Changing the volume affects the currently playing sound and any sounds played subsequently on that channel.

Note that you can set the volume of a non-existant channel and this will result in it being created. This allows you to set the volume *before* any sounds are played. If you attempt to get the volume of a non-existent channel, however, empty will be retuned.

To find out what sounds (if any) are currently playing and are scheduled for playing next on a given channel use:

mobileSoundOnChannel(*channel***)**

mobileNextSoundOnChannel(*channel***)**

These will return empty if no sound is currently (scheduled for) playing (or the channel doesn't exist).

To query a channel's current status use **mobileSoundChannelStatus**(). This returns one of the following:

- *stopped* – there is no sound currently playing, nor any sound scheduled to be playing
- *paused* – there are sounds scheduled to be played, but the channel is currently paused
- *playing* – a sound is currently playing on the channel

Managing Channels

To get a list of the sound channels that currently exist use:

mobileSoundChannels()

This returns a return-delimited list of the channel names.

Sound channels persist after any sounds have finished playing on them, retaining the last set volume setting. To remove a channel from memory completely use:

mobileDeleteSoundChannel *channel*

Sound channels only consume system resources when they are playing sounds, thus you don't need to be concerned about having many around at once (assuming most are inactive!).

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.

Font querying support

The list of available fonts can now be queried by using **the fontNames** function. This returns a return-delimited list of all the available font families.

The list of available styles can be queried by using the **fontStyles** function:

```
    fontStyles(fontFamily, 0)
```

This will return the list of all font names in the given family. It is these names which should be used as the value of **the textFont** property.

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.

Idle Timer configuration

By default, Android will dim the screen and eventually lock the device after periods of no user-interaction.

To control this behavior, use the following commands:

mobileLockIdleTimer

mobileUnlockIdleTimer

Locking the idle timer increments an internal lock count, while unlocking the idle timer decrements the lock count. When the lock count goes from 0 to 1, the idleTimer is turned off; when the lock count goes from 1 to 0, the idleTimer is turned on.

To determine whether the idleTimer is currently locked (i.e. turned off) use **mobileIdleTimerLocked()**.

Locale and system language query support

You can query the preferred language using the **mobilePreferredLanguages()** function. This returns a standard language tag (for example "en", "fr", "de", etc.)

You can query the currently configured locale using the **mobileCurrentLocale()** function. This returns a standard locale tag (for example "en_GB", "en_US", "fr_FR", etc.)

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.*

Clearing pending interactions

As interaction events (touch and mouse messages) are queued, it is possible for such messages to accumulate when they aren't needed. In particular, when executing 'waits', 'moves' or during card transitions.

To handle this case, the **mobileClearTouches** command has been added. At the point of calling, this will collect all pending touch interactions and remove them from the event queue.

Note that this also cancels any existing mouse or touch sequences, meaning that you (nor the engine) will not receive a mouseUp, mouseRelease, touchEnd or touchCancel message for any current interactions.

A good example of when this command might be useful is when playing an instructional sound:

```
on tellUserInstructions
    play specialFolderPath("engine") & slash & "Instruction_1.mp3"
    wait until the sound is "done"
    mobileClearTouches
end tellUserInstructions
```

Here, if the *mobileClearTouches* call was not made, any tough events the user created while the sound was playing would be queued and then be delivered immediately afterwards potentially causing unwanted effects.

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.*

Contact Access

Support to access and modify the Android contact list has been added as of LiveCode 5.5.1-rc-1. Interaction with the contact list can be controlled either via native user interfaces or directly from the LiveCode syntax.

UI Contact Access Features

Four native user interfaces are available that allow contacts to be created, picked, shown or updated.

Creating a Contact

You launch the native Android contact creation dialog by calling the command **mobileCreateContact**.

This allows the user to create a contact with the fields that the user considers to be required for the new contact.

The result of this command returns either "empty" if no contact was created or the ID of a successfully created contact.

Picking a Contact

The user can select a contact from the contact list by using the **mobilePickContact** command.

The user is presented with a contact list dialog that shows all the contacts in the contact list.

The result of this command returns either "empty" if no contact was selected or the ID of the selected contact.

Showing a Contact

It is possible to present the contact details of a contact to the user using the native Android contact viewer. You launch the contact viewer by calling the command:

mobileShowContact *contactID*

The dialog is only launched if the provided *contactID* exists in the contact list.

The result of this command returns either "empty" if no contact with the provided *contactID* exists, or the ID of the contact that was viewed.

Updating a Contact

A contact can be pre-populated with information before launching an Android contact creation dialog by using the command:

mobileUpdateContact *contactArray*

This feature allows interaction with the contact creation process to be streamlined for the user. If an application is already aware of some of the contact details that the user has to complete, then that data can be entered into the new contact automatically.

The information to pre-populate contact information is provided in form of an array. The array structure is detailed under heading "Contact Array Structure".

The result of this command returns either "empty" if no contact was created or the ID of a successfully created contact.

Syntax Contact Access Features

The LiveCode syntax supports direct contacts manipulation to create, find, remove a contact and to read contact data.

Contact Array Structure

The handlers **mobileUpdateContact, mobileAddContact** and **mobileGetContactData** all use a

common contact array format. The structure of the array is defined as follows:

Person Information

The contact's personal information is stored at the top level of the array and has the following keys.

- firstname - The first name.
- middlename - The middle name.
- lastname - The last name.
- alternatename - The alternative name.
- nickname - The nick name.
- firstnamephonetic - The phonetic transcription of the first name.
- middlenamephonetic - The phonetic transcription of the middle name.
- lastnamephonetic - The phonetic transcription of the last name.
- prefix - The name prefix.
- suffix - The name suffix.
- organization - The name of the organization.
- jobtitle - The job title.
- department - The name of the department.
- message - A person message.
- note - A person note.

E-Mail Addresses

The contact's email addresses are stored in subarrays under the key **email**. There are three categories of email address:

- home - The home e-mail address.
- work - The work e-mail address.
- other - An alternative e-mail address.

Each email address category is an integer indexed rray (starting at 1), allowing for a category to have any number of email addresses stored against it.

So for example, the contact's first home email address will be as follows:

tContactData[“email”][“home”][1]

Telephone Numbers

The contact's telephone numnbers are stored in subarrays under the key **phone**. There are ten categories of phone numbers::

- mobile - The mobile telephone number.
- main - The main telephone number.
- home – The home telephone number.

- work – The work telephone number.
- homefax - The home FAX number.
- workfax - The work FAX number.
- otherfax – An alternative FAX number (iOS 5.0 and later).
- pager - The pager number.
- other - An alternative telephone number.

Each phone number category is an integer indexed array (starting at 1), allowing for a category to have any number of phone numbers stored against it.

So for example, the contact's first mobile phone number will be as follows:

tContactData[“phone”][“mobile”][1]

Addresses

A contact's addresses are stored as subarrays under the key **address.** There are three categories of address:

- home – The home address.
- work – The work address.
- other – An alternative address.

Each address category is an integer indexed array (starting at 1), allowing for a category to have any number of addresses stored against it.

Each individual address has the following keys:

- street - The address's street.
- city - The address's city.
- state - The address's state.
- zip - The address's ZIP code.
- country - The address's country.
- countrycode – The address's country code.

So, for example, the street of the contact's first home address is as follows:

tContactData[“address”][“home”][1][“street”]

Adding a Contact

You can add a contact by calling the command **mobileAddContact**. This allows you to populate the entries of the new contact record with token,value pair strings of the following form:

mobileAddContact *contactArray*

The contact array structure is detailed under heading "Contact Array Structure".

The result of this command returns either "empty" if no contact was created or the ID of a successfully created contact.

Finding a Contact

The contact list database can be queried, based on the contact name, using the command:

mobileFindContact *contactName*

It is possible to provide parts of the contact's name that is to be queried as the argument *contactName*. The first letter of the given name or surname would be sufficient to provide a search for a given contact.

The result of this command returns either "empty" if no contact could be found or a comma delimited list of IDs of the contacts that match the search.

Removing a Contact

A contact can be removed from the contact list by using the command:

mobileRemoveContact *contactID*

The result of this command returns either "empty" if no contact with the provided ID could be found or the ID of the contact that was deleted.

Getting Contact Data

Information stored against a particular contact can be retrieved by calling the function:

mobileGetContactData(*contactID*)

This function extracts all of the contact fields that are supported in LiveCode and returns them in form of an array with the array keys representing the tokens and the corresponding array values representing the contact specific information.

The contact array structure is detailed under heading "Contact Array Structure".

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

Local notifications

Local notifications allow applications to schedule notifications with the operating system. The notification can be received when the application is running in the foreground, the application is running in the background or the application is not running at all. The notification is delivered differently, depending on the mode in which the application is in at the time the notification is received.

mobileCreateLocalNotification *alertBody*, *alertTitle*, *alertPayload*, *alertTime*, *playSound*, *[badgeValue]*

Use the command **mobileCreateLocalNotification** to schedule a notification with the OS.

- *alertBody*- the text that is to be displayed on the status bar entry that is raised when the application is not running
- *alertTitle* - the status bar title text that appears when the application is not running
- *alertPayload* - a text payload that can be sent with the notification request. This payload is presented to the user via the **localNotificationReceived** message
- *alertTime* - the time at which the alert is to be sent to the application
- *playSound* - boolean to indicate if a sound is to be played when the alert is received
- *badgeValue* - the number value to which the status bar logo is to be set.

A return delimited list of all the currently pending notifications can be fetched using the function:

mobileGetRegisteredNotifications().

You can query a given notification using the function **mobileGetNotificationDetails.**

mobileGetNotificationDetails(*notification***)**

This returns an array with the following entries:

- **body** - status bar entry when the application is not running
- **title** - the title of the status bar entry
- **payload** - the text presented to the app via the **localNotificationReceived** message
- **play sound** - boolean indicating if a sound is to be played when the notification is received.
- **badge value** - the number displayed on the status bar icon when the notification is received. No number will be displayed if this is zero

To cancel a notification use command **mobileCancelLocalNotification.**

mobileCancelLocalNotification *notification*

Here, the *notification* parameter is a value returned by **mobileGetRegisteredNotifications().**

To cancel all pending notifications, use the command **mobileCancelAllLocalNotifications.**

When you app receives a notification, the message **localNotificationReceived** will be sent.

localNotificationReceived *message*

Here, the *message* parameter is the payload specified when the notification was created.

If your application is not running when a notification is received, an entry in the Android notification centre will be created. If a badge number is received with the notification, then that badge value is displayed on the notification centre icon, but only if the application is not currently running. When your application is started up, if the notification centre message has not been dismissed, the **localNotificationReceived** message will be sent on startup.

If the application is running when a notification is received, **localNotificationReceived** will be sent.

Push notifications

Push notifications allow apps to avoid frequently polling for the availability of new remote data by providing a mechanism whereby notifications can be sent to the mobile device.

Yo must first register with the Google's C2DM service (http://code.google.com/android/c2dm/signup.html). Once you have successfully signed up for the service, you must include your account name in the Standalone Builder.

The **pushNotificationRegistered** message is sent once the application starts up and registers with the Push Notification Server.

pushNotificationRegistered *signature*

The *signature* parameter is the signature of the device. This is the unique device's signature that the Push Notification Server uses in order to send a notification to the device. This can be fetched at any point using the function **mobileGetDeviceToken().**

The application only tries to register with the Push Notification Server if the application was configured to handle Push Notifications in the Standalone application Builder.

The **pushNotificationReceived** message is sent once the application receives a push notification from a Push Notification Server.

pushNotificationReceived *message*

Here, the message parameter is the payload specified when the notification was created.

If your application is not running when a notification is received, an entry in the Android notification centre will be created. If a badge number is received with the notification, then that badge value is displayed on the notification centre icon, but only if the application is not currently running. When your application is started up, if the notification centre message has not been dismissed, the **pushNotificationReceived** message will be sent on startup.

If the application is running when a notification is received, **pushNotificationReceived** will be sent.

The **pushNotificationRegistrationError** message is handled once the application starts up and tried, but failed to register with the Push Notification Server.

pushNotificationRegistrationError *errorMessage*

When receiving a push notification the engine processes the following data fields:

- data.body - the message body displayed in the status bar (default: "User interaction requested")
- data.title - the title of the message displayed in the status bar (default: the app label)
- data.badge_value - the badged number to display along with the status-bar message
- data.play_sound - (true / false) indicates whether or not a sound should be played when the notification is received
- data.payload - the message that will be delivered to the app in its remoteAlert handler

Custom URL schemes

Specifying a custom URL allows you app to be woken up when the given URL is invoked.

To specify a custom URL, add the desired URL to the "URL Name" field of the standalone application builder. For example, if you specify "myURL" as the URL name, then when the URL myURL:// is invoked, if installed, your app will be woken.

Extra parameters can be passed in the URL in the following format:

myURL://

myURL://some/path/here

myURL://?foo=1&bar=2

myURL://some/path/here?foo=1&bar=2

If you app is woken by a custom URL, the message **urlWakeUp** will be sent to the current card.

urlWakeUp *urlString*

A single parameter will be passed detailing the URL used to launch your app. This value can bet retrieved at any point using the function **mobileGetLaunchURL()**. If the app was not launched from a URL then this will return empty.

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

In App Advertising

Ads are supplied by our ad partner inneractive and come in three different types: banner, full screen and text. Before you can begin placing ads, you must first register your app with inneractive.

To do this, sign up with inneractive at the following URL: http://runrev.com/store/account/inneractive/. Once successfully signed up with inneractive you must generate a key for your app. Do this by clicking on the "Add App" tab of the inneractive dashboard and following the instructions provided.

Once you have a key for your app, you must register this with LiveCode using the **mobileAdRegister** command. You will now be ready to place ads, using the **mobileAdCreate** command.

Registering Your App Key

Before you can begin creating ads, you must first register your app's unique Interactive identifier. All ad activity, including any revenue generated, will be logged against this id.

mobileAdRegister *appKey*

Creating & Managing Ads

Once your app key has been registered, you are now ready to create an ad. To do so, use the command **mobileAdCreate.**

mobileAdCreate *ad*, [*type*], [*topLeft*], [*metaData*]

The parameters are as follows:

- *type*: The type of ad. One of "banner", "text" or "full screen". Defaults to "banner".
- *name*: The name of the ad to create. This will be used to reference the ad throughout its lifetime.
- *topLeft*: The location in pixels of the top left corner of the ad. Defaults to 0,0.
- *metaData*: An array of values that will be used to target the ad. The keys are as follows:
    - **refresh**: A value in seconds defining how often the ad will refresh, between 30 and 300. Defaults to 120.
    - **age**: An integer defining the expected age of the target user.
    - **gender**: The expected gender of the target user. The allowed values are M, m F, f, Male, Female.
    - **distribution id**: The distribution Channel ID (559 for banner ads and full screen ads, 600 for text ads).
    - **phone number**: The user's mobile number (MSISDN format, with international prefix).
    - **keywords**: Keywords relevant to this user's specific session (comma separated, without spaces).

The ad support and internet permissions checkboesx must be ticked in the Standalone Application Settings. It is also recommended that both the fine and coarse location checkboxes are ticked. This allows the ad served to be tailored to the user's location.

Ads can be deleted at any time using to command **mobileAdDelete**.

mobileAdDelete *ad*

You can get and set the top left of an ad using the following:

mobileAdGetTopLeft(*ad)*

mobileAdSetTopLeft(*ad, topLeft)*

The top left is the pixel coordinates of the top left corner of the ad.

You can get and set the visibility of an ad using the following:

mobileAdGetTopVisible(*ad)*

mobileAdSetTopVisible(*ad, visible)*

The visible is a boolean, set to true if the ad is visible, false otherwise.

A list of all the currently active ads can be fetched using the function:

mobileAds()

This returns a return-delimited list of the ad names.

Messages

When an add is loaded or refreshed, the message **adLoaded** will be sent to the current card.

adLoaded *default*

Here, *default* is a boolean, set to true if the loaded ad is a default ad.

If a user clicks on an ad, the **adClicked** message will be sent to the current card.

adClicked

If an ad fails to load the **adLoadFailed** message will be sent to the current card.

adLoadFailed

If an ad is about to resize the **adResizeStart** message will be sent to the current card.

adResizeStart

When an ad has finished resizing the **adResizeEnd** message will be sent to the current card.

adResizeEnd

If an ad is about to expand the **adExpandStart** message will be sent to the current card.

adExpandStart

When an ad has finished expanding the **adExpandEnd** message will be sent to the current card.

adExpandEnd

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

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

Busy indicator

Use the command **mobileBusyIndicatorStart** to display an activity dialog that will sit above all other controls and block user interaction.

mobileBusyIndicatorStart *style, [label]*

The *style* parameter is used to determine the display style of the dialog. At the moment, only "square" is supported. This creates a square dialog box containing an animated progress indicator and an optional label.

The optional *label* parameter is used to pass any text which you wish to be displayed in the dialog.

To dismiss the dialog, use the **mobileBusyIndicatorStop** command.

mobileBusyIndicatorStop

Modal Pick-Wheel support

You can present the user with a list of choices to pick from using standard Android interface elements using:

mobilePick *optionList, initialIndex, [style], [button]*

Where *optionList* is a return-delimited list to choose from, and *initialIndex* is the (1-based) index of the item to be initially highlighted. The item the user chooses is returned in **the result**.

The *initialIndex* is the (1-based) index of the item to be initially highlighted.

The optional *style* parameter determines the type of display used If equal to "checkmark" a check-mark (radio button) will be put against the currently selected item.

The optional button parameter specifies if "Cancel" and/or "Done" buttons should be forced to be displayed with the picker dialog.

- cancel - display the Cancel button on the Picker
- done - display the Done button on the Picker
- cancelDone - display the Cancel and Done buttons on the Picker

If the 'Cancel' button is displayed, then any selection made by the user can be canceled, the result contains the initial index.

If the 'Done' button is displayed, then the result contains the initial index.

Pressing the back key has the same result as the 'Cancel' button.

Media picker support

You can present the user with the standard Android media picker using:

mobilePickMedia

A return separated list of all the media items the user has picked will be present in the result. A media item can be played back using the **play** command.

Date picker support

You can present the user with a standard Android date picker using:

mobilePickDate *[mode], [initial], [min], [max]*

The mode parameter determines the mode of the date picker and can be one of the following:

- *date*
- *time*

The mode defaults to date.

The initial parameter determines the initial date to be displayed by the date picker. If this is empty, the current date will be used. This should be a time in seconds since the Unix Epoch.

The min parameter is the start range of the date picker. If this value is empty, there is no lower boundary. The value is ignored if min is greater than max. This should be a time in seconds since the Unix Epoch.

The max parameter is the end range of the date picker. If this value is empty, there is no upper boundary. The value is ignored if max is less than min. This should be a time in seconds since the Unix Epoch.

When the date picker is dismissed by the user, the selected date will be stored in the result.

Native Controls

Low-level support has been added for creating and manipulating some native controls.

To create a native control use:

mobileControlCreate *controlType,* [*name*]

Where *controlType* is the type of control you wish to create – "browser" and *name* is an optional string to use to identify the control in the other functions. The *name* must be unique amongst all existing controls and cannot be an integer. The unique (numeric) id for the new control is returned in **the result**.

To destroy a native control use:

mobileControlDelete *idOrName*

Where *idOrName* is the numeric id returned by *mobileControlCreate,* or the *name* of the control if provided.

A list of all native controls currently in existence can be fetched using the **mobileControls()** function. This returns a return-delimited list of control names or ids. Where a control has a name that is used, otherwise its id is used.

Once such a control has been created, you can configure it using:

mobileControlSet *idOrName*, *property*, *value*

Where

- *idOrName* is the numeric id returned by *mobileControlCreate,* or the name of the control if provided.
- *property* is the name of the property to change
- *value* is the value of the property to change to

Properties can also be read by using **mobileControlGet**(*id, property).*

Control specific behavior can be invoked by using:

mobileControlDo *idOrName, action, ...*

Where *action* is what is to be done, and the parameters are action/control specific.

While in the context of a message that has been dispatched from a native control, you can use the **mobileControlTarget()** function to fetch the name (or id, if no name is set) of the control that sent the message.

In general, any messages dispatched by the native control will be sent to the object containing the script which created it, this also works correctly with behaviors – messages being sent to the object referring to the behavior, and not the behavior's object.

All controls

All native Android controls have a common set of properties and actions.

Properties

id	read-only	The unique (integer) id of the control.
name	read-only	The unique name of the control if one was provided at creation time, empty otherwise.
rect	read/write	The bounds of the control, relative to the top-left of the card.
visible	read/write	Set to true or false to determine whether the control should be displayed.

Browser control

A native Android browser control is created using a control type of "browser".

Properties

url	read/write	The url to be loaded into the web-view.
canAdvance	read-only	Returns true if there is a next page in the history (wraps the

		canGoForward property of UIWebView).
canRetreat	read-only	Returns true if there is a previous page in the history (wraps the *canGoBack* property of UIWebView).
scrollingEnabled	read/write	Whether or not the browser can be scrolled (boolean).
canBounce	read/write	Determines whether the scroller will 'bounce' when it hits the edge of the contentRect (boolean)

Actions

mobileControlDo *id,* "advance"

Move forward through the history (wraps the *goForward* method of UIWebView).

mobileControlDo *id,* "retreat"

Move backward through the history (wraps the *goBack* method of UIWebView).

mobileControlDo *id,* "reload"

Reload the current page (wraps the *reload* method of UIWebView).

mobileControlDo *id,* "stop"

Stop loading the current page (wraps the *stopLoading* method of UIWebView).

mobileControlDo *id,* "load", *baseUrl, htmlText*

Loads as page consisting of the given *htmlText* with the given *baseUrl* (wraps the *loadHtmlString* method of UIWebView).

mobileControlDo *id,* "execute", *script*

Evaluates the given JavaScript *script* in the context of the current page (wraps the *stringByEvaluationJavaScriptFromString* method of UIWebView).

Messages

browserStartedLoading *url*

Sent when the given url has started to load (sent in response to the *webViewDidFinishLoad* delegate method).

browserFinishedLoading *url*

Sent when the given url has finished loading (sent in response to the *webViewDidStartLoad* delegate method).

browserLoadFailed *url*, *error*

Sent when the given url fails to load (sent in response to the *webView:didFailLoadWithError:* delegate method).

Scroller control

A native Android browser control is created using a control type of "scroller".

Rather than act as a container for other controls, the 'scroller' is intended to be used as an overlay on part of the screen you wish to interact with the proper Android scrollbars. By responding to the

various scroller messages, you can move LiveCode controls or set the appropriate scroll properties of group and fields to get a native scrolling effect.

Properties

contentRect	read/write	The rectangle over which the scroller scrolls. This is distinct from the scroller's rect, and is essentially the minimum/maximum values of the scroll properties (adjusted for the size of the scroller). This is a comma-separated list of four integers, describing a rectangle.
hScroll	read/write	The horizontal scroll offset. This is an integer value ranging between the left and right of the contentRect, adjusting appropriately for the size of the scroller (i.e. contentRect.left to contectRect.right – rect.width).
vScroll	read/write	The vertical scroll offset. This is an integer value ranging between the top and bottom of the contentRect, adjusting appropriately for the size of the scroller (i.e. contentRect.top to contectRect.bottom – rect.height).
scrollingEnabled	read/write	Determines whether touches on the scroller cause scrolling (maps to the UIScrollView *scrollEnabled* property). This is a boolean value.
hIndicator	read/write	Determines whether the horizontal indicator should be displayed when scrolling. This is a boolean value.
vIndicator	read/write	Determines whether the vertical indicator should be displayed when scrolling. This is a boolean value.
tracking	read-only	Returns true if the scroller is monitoring a touch for the start of a scroll action. This is a boolean value.
dragging	read-only	Returns true if the scroller is currently performing a scroll action. This is a boolean value.

Messages

scrollerBeginDrag

This message is sent when a scroll initiating drag is started.

scrollerEndDrag *didDecelerate*

This message is sent when a scroll initiating drag is finished.

scrollerDidScroll *hScroll, vScroll*

This message is sent when the scroll properties of the scroller have changed.

Player control

A native Android media player is created using a control type of "player".

Properties

filename	read/write	The filename of URL of the media to play. Setting the filename of the player automatically 'prepares' the movie for playback.
showController	read/write	Determines whether the controller will be displayed over the content. This is a boolean value.
looping	read/write	Determines whether the playback of the movie should loop indefinitely. This is a boolean value.
currentTime	read/write	The current position of the playhead, measured in milliseconds (maps to the native currentPlaybackTime property). This is an integer value.

Actions

mobileControlDo *id,* "play"

Start playing the content of the player.

mobileControlDo *id,* "pause"

Pause the content at the current position.

mobileControlDo *id,* "stop"

Stop playing the content of the player.

Messages

playerFinished

The content has finished playing through.

playerStopped

The content finished playing through due to a user exit.

playerError

The content finished playing due to an error.

Input control

A native text field control is created using a control type of "input".

The input control allows the editing of a single line of text, with the 'return' key ending editing and

allowing the application to perform an appropriate action.

Properties

text	read/write	The content of the control (maps to the native text property). This is a string value.
unicodeText	read/write	The content of control encoded as UTF-16 (maps to the native text property). This is a binary value.
textColor	read/write	The color to use for the text in control (maps to the native textColor property). This is either a standard color name, or a string of the form *red,green,blue* or *red,green,blue,alpha*. Where the components are integers in the range 0 to 255.
fontSize	read/write	The size of the font to use for text in the control. This is an integer value.
textAlign	read/write	The alignment to use for text in the control (maps to the native textAlignment property). This is one of *left*, *center* or *right*.
autoCapitalizationType	read/write	Determines when the shift-key is automatically enabled (maps to the native autocapitalizationType property). This is one of the following: • *none* – the shift-key is never automatically enabled • *words* – the shift-key is enabled at the start of words • *sentences* – the shift-key is enabled at the start of sentences • *all characters* – the shift-key is enabled at the start of each character
autoCorrectionType	read/write	Determines whether auto-correct behavior should be enabled (maps to the native autocorrectionType property). This is one of the following: • *default* – use the appropraite auto-correct behavior for the current script system. • *no* – disable auto-correct behavior • *yes* – enable auto-correct behavior
keyboardType	read/write	Determines what kind of keyboard should be displayed (maps to the native keyboardType property). This is one of the following:

- *default* – the normal keyboard
- *alphabet* – the alphabetic keyboard
- *numeric* – the numeric keyboard with punctuation
- *url* – the url entry keyboard
- *number* – the number pad keyboard
- *phone* – the phone number pad keyboard
- *contact* – the phone contact pad keyboard
- *email* – the email keyboard
- *decimal* – the decimal numeric pad keyboard (iOS 4.1+)

returnKeyType	read/write	Determines what kind of return-key the keyboard should have (maps to the native returnKeyType property). This is one of the following: • *default* – the normal return key • *go* – the 'Go' return key • *next* – the 'Next' return key • *search* – the 'Seach' return key • *send* – the 'Send' return key • *done* – the 'Done' return key
contentType	read/write	Determines what kind of content the control contains. This is one of the following: • *plain* – plain, unstyled text • *password* – plain text displayed in the standard iOS password style.
editable	read/write	Whether the field contents can be edited (boolean)
dataDetectorTypes	read/write	Determines what types of data should be detected and automatically converted to clikckable URLs. This is a comma delimited list of none or more of the following: • phone number • link • address • email
selectedRange	read/write	Start & end of current selection (two comma separated numbers).
scrollingEnabled	read/write	Whether or not the field can be scrolled (boolean).
multiline	read/write	If true, this field can contain multiple lines of text, wraps text to fit horizontally, and scrolls vertically. If false, the field contains a

		single line of text which can scroll horizontally (boolean).
verticalTextAlign	read/write	Determine the vertical alignment of the text within the field. One of “top”, “center” or “bottom”. (Multi-line controls only).

Messages

inputBeginEditing

The control has become focused and editing has commenced.

inputEndEditing

The control has lost focus and editing has ceased.

inputTextChanged

An editing operation has taken place and the content of the control has changed.

inputReturnKey

The return key has been pressed and focus removed from the input control.

Multi-line Input control

A multi-line field control is created using a control type of "multiline". This behaves in the same manner as the single line input control except that the field can contain multiple lines of text, wraps text to fit horizontally, and scrolls vertically

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

Contact access updates (6.0-dp-3)

The data structures handled by the contact access handlers has been unified to use a common contact array.

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.
5.0.2-dp-1 (2011-11-25)	MM	No changes.

5.0.2-rc-1 (2011-12-02)	MM	No changes.
5.0.2-gm-1 (2011-12-12)	MM	Fixed bug – apps that use parent scripts crash on restart. Fixed bug – certain visual effects cause black flash when using OpenGL compositor (9901). Fixed bug - pause at end of visual effect on Android due to missing redraw (9907). Fixed bug - flash of previous frame at end of effect on (9908).
5.5.0-dp-1 (2011-02-13)	MM	Added support for native browser control. Added modal pick wheel support. Added date picker support. Added support for location, heading, rotation sensors. Added support for multi-channel sound. Added support for busy inidicators. Fixed bug – post to URL does not work (9964). Fixed bug – HTTP basic authentication not supported.
5.5.0-dp-2 (2011-02-27)	MM	Added support for local notifications. Added support for push notifications. Added support for custom URL schemes. Added support for in app advertising. Added support for vibrate. Added support for beep. Added support for text messaging. Fixed bug - pickers do not return 0 on cancel (9999).
5.5.0-dp-3 (2011-03-09)	MM	Added support for custom fonts. Added support for Idle Timer configuration. Added localisation calls. Fixed bug - list bullets don't render.
5.5.0-rc-1 (2012-03-14)	MM	Removed support for in app advertising. Fixed bug – location sensor does not restart after being stopped (10077). Fixed bug - if no reading has been retrieved from Android sensor, fetching a reading should return empty.
5.5.0-rc-2 (2012-03-16)	MM	Fixed bug - not specifying a type crashes mobilePickDate.
5.5.0-gm-1 (2012-03-20)	MM	Fixed bug - apps freeze on restart if heading has not been stopped in previous session (10104).
5.5.0-gm-2 (2012-03-23)	MM	Fixed bug – keyboard switches back from numeric to alphabetic after a number has been entered (10040).
5.5.0-gm-3 (2012-03-26)	MM	No changes.
5.5.1-dp-1 (2012-04-05)	MM	Re-instated support for in-app advertising. Added support for mdeia picker. Added support for saving images to the user's photo album. Added support for native scroller control. Added support for native player control.
5.5.1-dp-2 (2012-05-04)	MM	Fixed bug – mobileAdCreate no longer requires all parameters to be specified. Updated ad support to use latest inneractive APIs.
5.5.1-rc-1 (2012-05-10)	MM	Added support for address book access. Added support for native field controls.
5.5.1-rc-2 (2012-05-11)	MM	Added address book support for phonehome and phonework.

		Removed support for phonemain.
5.5.1-rc-3 (2012-06-01)	MM	Fixed bug - mobile controls can remain in list after deleted (10203). Fixed bug - mobile pick crashes Android simulator (10211). Fixed bug – get URL does not return error code in result and error page delivered by server (10243).
5.5.1-rc-4 (2012-06-15)	MM	Enabled plugins in native browser control. Fixed bug – crash on certain visual effects (10247).
5.5.1-gm-1 (2012-06-19)	MM	Updated ad support to use latest inneractive ad APIs.
5.5.2-dp-1 (2012-08-17)	MM	Updated ad support to use latest inneractive ad APIs. Fixed bug – mobileLockTimer fails (10316). Fixed bug - scrolling large images creates vertical chunks of cd color over the image (10285). Updated engine to support user created externals.
5.5.2-rc-1 (2012-08-31)	MM	Fixed bug – changing contentRect does not update scroller (10318).
5.5.2-rc-2 (2012-09-07)	MM	Fixed bug – text does not render within tight loops (10333).
5.5.2-gm-1 (2012-09-11)	MM	No changes.
5.5.2-gm-2 (2012-09-13)	MM	No changes.
5.5.3-rc-1 (2012-09-26)	MM	Fixed bug - local files cannot be opened in native browser (10365). Fixed bug - setting the fileName of an image does not work (10394). Added two new browser properties canBounce and scrollingEnabled (enhancement request 10304).
5.5.3-rc-2 (2012-10-09)	MM	Fixed bug – native text field does not dismiss keyboard (10219). Fixed bug – restarting app causes hang (10439). Added function mobilePixelDensity. Added keyboardActivated/keyboardDeactivated messages. Updated the working screenRect to take into account space occupied by keyboard.
5.5.3-rc-3 (2012-10-19)	MM	Fixed bug – multi channel audio playback clipped on Kindle Fire (10437). Fixed bug – numeric keyboard input not appearing (10466).
5.5.3-gm-1 (2012-10-25)	MM	Fixed bug – tel: and mailto: links do not work in browser control (10440). Fixed bug – mail always returns cancel (10486). Reverted the working screenRect to taking only the status bar into account. Added the effective working screenRect to take into account space occupied by keyboard.
5.5.3-gm-2 (2012-10-31)	MM	No changes.
6.0.0-dp-1 (2012-11-08)	MM	Fixed bug – date and time returned by date picker is offset by the locale of the device (10483). Fixed bug – movieTouched message not sent (10510). Added property “verticalTextAlign” to native fields.
6.0.0-dp-2 (2012-11-26)	MM	Fixed bug – stretching certain images can cause artefacts (bug 10500).
6.0.0-dp-3 (2012-12-11)	MM	Updated the address book features to use a common contact array

		structure (bugs 10347, 10544). Fixed bug – native controls are not created when an app is restarted.
6.0.0-dp-4 (2012-12-20)	MM	No changes.
6.0.0-dp-5 (2013-03-01)	MM	Added optional opacity parameter to busy indicator (bug 10642). Fixed bug – playRate has no effect on native player (10632). Fixed bug – preceding or trailing whitespace not ignored in URLs (10030). Fixed bug - keyDown/keyUp messages now sent correctly on JellyBean (10684). Fixed bug – certain paths resolved incorrectly (10636). Fixed bug - temporary file not always writable by the camera app (10482).
6.0.0-rc-1 (2013-03-08)	MM	No changes.
6.0.0-rc-2 (2013-03-15)	MM	Fixed bug – crash when calling replaceText in preOpenStack (10713).
6.0.0-rc-3 (2013-03-25)	MM	Fixed bug – mobileSetKeyboardType has no effect on ask dialogs (10758).
6.0.0-rc-4 (2013-04-02)	MM	No changes.
6.0.0-rc-5 (2013-04-04)	MM	No changes.
6.0.0-rc-6 (2013-04-05)	MM	No changes.
6.0.0-rc-7 (2013-04-05)	BB	No changes.
6.0.0-gm-1 (2013-04-09)	MM	No changes.
6.0.1-rc-1 (2013-04-19)	MM	No changes.
6.0.1-gm-1 (2013-04-30)	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.
5.0.2-dp-1 (2011-11-25)	MM	No changes.
5.0.2-rc-1 (2011-12-02)	MM	No changes.
5.0.2-gm-1 (2011-12-12)	MM	No changes.
5.5.0-dp-1 (2011-02-13)	MM	Updated Standalone Builder to add GPS permissions.
5.5.0-dp-2 (2011-02-27)	MM	Updated Standalone Builder to include support for vibrate, notifications and a default icon.
5.5.0-dp-3 (2011-03-09)	MM	Added support for .ttf and .ttc font files in the Copy Files pane of the standalone builder.

		Updated Standalone Builder to include support for idle timer. Fixed bug - minimum version not honoured in Standalone Settings (1002).
5.5.0-rc-1 (2012-03-14)	MM	No changes.
5.5.0-rc-2 (2012-03-16)	MM	No changes.
5.5.0-gm-1 (2012-03-20)	MM	Fixed bug – error building applications when "copy files" includes nested folders.
5.5.0-gm-2 (2012-03-23)	MM	No changes.
5.5.0-gm-3 (2012-03-26)	MM	No changes.
5.5.1-dp-1 (2012-04-05)	MM	No changes.
5.5.1-dp-2 (2012-05-04)	MM	Added "Ad Support" check box to standalone builder.
5.5.1-rc-1 (2012-05-10)	MM	Added contact support permissions to standalone builder.
5.5.1-rc-2 (2012-05-11)	MM	Added missing XML to package.
5.5.1-rc-3 (2012-06-01)	MM	No changes.
5.5.1-rc-4 (2012-06-15)	MM	No changes.
5.5.1-gm-1 (2012-06-19)	MM	No changes.
5.5.2-dp-1 (2012-08-17)	MM	Updated the deployment process to support development on Linux. Added support for user created externals.
5.5.2-rc-1 (2012-08-31)	MM	No changes.
5.5.2-rc-2 (2012-09-07)	MM	Fixed bug – files within sub-folders are not copied in apk correctly (10345).
5.5.2-gm-1 (2012-09-11)	MM	No changes.
5.5.2-gm-2 (2012-09-13)	MM	No changes.
5.5.3-rc-1 (2012-09-26)	MM	No changes.
5.5.3-rc-2 (2012-10-09)	MM	No changes.
5.5.3-rc-3 (2012-10-19)	MM	No changes.
5.5.3-gm-1 (2012-10-25)	MM	No changes.
5.5.3-gm-2 (2012-10-31)	MM	No changes.
6.0.0-dp-1 (2012-11-08)	MM	No changes.

6.0.0-dp-2 (2012-11-)	MM	No changes.
6.0.0-dp-3 (2012-12-11)	MM	No changes.
6.0.0-dp-4 (2012-12-20)	MM	No changes.
6.0.0-dp-5 (2013-03-01)	MM	No changes.
6.0.0-rc-1 (2013-03-08)	MM	No changes.
6.0.0-rc-2 (2013-03-15)	MM	No changes.
6.0.0-rc-3 (2013-03-25)	MM	No changes.
6.0.0-rc-4 (2013-04-02)	MM	No changes.
6.0.0-rc-5 (2013-04-04)	MM	No changes.
6.0.0-rc-6 (2013-04-05)	MM	No changes.
6.0.0-rc-7 (2013-04-08)	BB	No changes.
6.0.0-gm-1 (2013-04-09)	MM	No changes.
6.0.1-rc-1 (2013-04-19)	MM	No changes.
6.0.1-gm-1 (2013-04-30)	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.
Revision 38 (2011-11-25)	MM	No changes.
Revision 39 (2011-12-02)	MM	No changes.
Revision 40 (2011-12-12)	MM	Updated section “Non-file URL access”.
Revision 41 (2012-02-13)	MM	Added section “Native controls” Added section “Modal pick wheel”. Added section “Date picker”. Added section “Sensor tracking” Added section “Heading tracking”. Added section “Location tracking”. Updated section “Accelerometer support”. Added section “Multi-Channel sound support”. Added section “Busy Indicator”.
Revision 42 (2012-02-27)	MM	Added section “Local Notifications”. Added section “Push Notifications”. Added section “Custom URL Schemes”. Added section “In App Advertising”. Added section “System Alert Support”. Added section “Vibration support”. Added section “Text Messaging”.
Revision 43 (2012-03-09)	MM	Added section “Adding custom fonts”

		Updated section “What doesn't work”.
		Added section “Font querying support”.
		Added section “Idle timer configuration”.
		Added section “Locale and system language query support”.
Revision 44 (2012-03-14)	MM	Removed section “In App Advertising”.
Revision 45 (2012-03-16)	MM	No changes.
Revision 46 (2012-03-20)	MM	No changes.
Revision 47 (2012-03-23)	MM	No changes.
Revision 48 (2012-03-26)	MM	No changes.
Revision 49 (2012-04-05)	MM	Added section “Saving photos to the user's album”.
		Added section “Media picker support”.
		Added section “Scroller control”.
		Added section “Player control”.
		Added section “In App Advertising”.
Revision 50 (2012-05-04)	MM	Updated section “In App Advertising”.
Revision 51 (2012-05-10)	MM	Added section “Contact Access”.
		Added section “Input control”
		Added section “Multi-line input control”.
Revision 52 (2012-05-10)	MM	Updated section “Contact Access”.
Revision 53 (2012-06-01)	MM	Updated section “Contact Access”.
Revision 54 (2012-06-01)	MM	No changes.
Revision 55 (2012-06-19)	MM	Updated section “In App Advertising”.
Revision 56 (2012-08-17)	MM	Updated section “Overview”.
		Updated section “Getting Started”.
		Updated section “Prerequisites”.
		Updated section “Configuring LiveCode”.
Revision 57 (2012-08-31)	MM	No changes.
Revision 58 (2012-09-07)	MM	No changes.
Revision 59 (2012-09-11)	MM	No changes.
Revision 60 (2012-09-13)	MM	No changes.
Revision 61 (2012-09-26)	MM	Updated section “Browser Control”.
Revision 62 (2012-10-09)	MM	Added section “Activation Notifications”.
		Updated section “Resolution Handling”.
Revision 63 (2012-10-19)	MM	Updated section “System Dialogs – answer and ask”.
		Added section “Clearing pending interactions”.
Revision 64 (2012-10-24)	MM	Updated section “Email Composition”.
		Updated section “Resolution Handling”.
Revision 65 (2012-10-31)	MM	No changes.
Revision 66 (2012-11-08)	MM	Updated section “Input Control”.
Revision 67 (2012-11-26)	MM	No changes.
Revision 68 (2012-12-11)	MM	Updated section “Noteworthy Changes”.
		Updated section “Contact Access”
		Updated section “Push Notifications”.
Revision 69 (2012-12-20)	MM	No changes.
Revision 70 (2013-03-01)	MM	Updated section “Sensor Tracking”.
Revision 71 (2013-03-08)	MM	No changes.
Revision 72 (2013-03-15)	MM	No changes.
Revision 73 (2013-03-25)	MM	No changes.
Revision 74 (2013-04-02)	MM	No changes.

Revision 75 (2013-04-04)	MM	No changes.
Revision 76 (2013-04-05)	MM	No changes.
Revision 77 (2013-04-08)	BB	No changes.
Revision 78 (2013-04-09)	MM	No changes.
Revision 79 (2013-04-19)	MM	No changes.
Revision 80 (2013-04-30)	MM	No changes.