Android Deployment Release Notes (4.6.1)

Table of Contents

Overview......3
Getting Started......4
Prerequisites......4
Configuring LiveCode......4
Configuring an Android standalone......5
Configuring an emulated device......5
Configuring a real device......6
Testing an Android application......6
A First Project......7
Configuring an Android Application......8
Setting manifest options......8
Adding a launcher icon......8
Adding a splash image (personal and educational)......8
Adding a default launch image (trial)......9
Deployment Features......10
Standalone builder messages......10
General Engine Features......11
Engine version......11
What doesn't work......11
What does work......11
Windowing and Stacks......12
System Dialogs – answer and ask......12
Non-file URL access......12
Externals......14
Android Specific Engine Features......15
Limitations......15
Multi-touch events......15
Mouse events......15
Motion events......16
Hardware back key support......16
Accelerometer support......16
Photo album and camera support......17
Taking or choosing photos......17
Keyboard Input......17
Configuring keyboard type......17
Orientation handling......18
Auto-rotation support......18
Querying orientation......18
Controlling auto-rotation......18
Orientation changed notification......19
Resolution handling......19
Email composition......19
Basic support......19

Advanced support.. 20
File and folder handling...21
Basic sound playback support...22
URL launching support..23
Hardware and system version query support..23
Querying camera capabilities...23
Change Logs and History..25
Engine Change History...25
Deployment Change History.. 25
Document Change History...25

Overview

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

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

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

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

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

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

Getting Started

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

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

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

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

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

Prerequisites

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

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

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

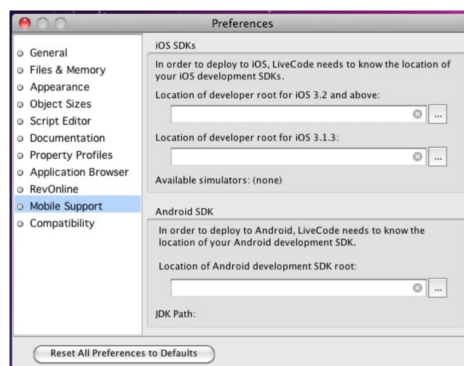
- Mac OS 10.5.x or later
- LiveCode 4.5.3 or later
- The Android SDK: http://developer.android.com/sdk/index.html

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

Configuring LiveCode

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

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

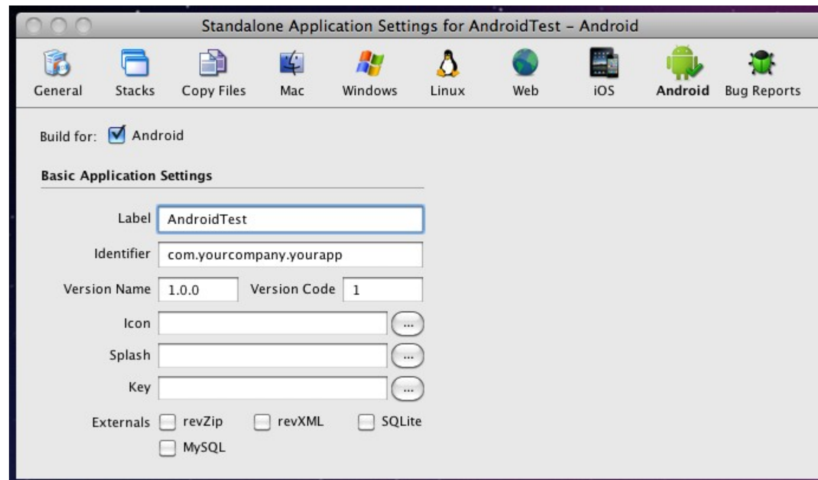


Use this pane to choose the correct SDK paths by clicking the '…' button after the *Android development SDK location* field.

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

Configuring an Android standalone

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



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

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

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

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

Configuring an emulated device

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

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

- Make sure the SDK Manager is running.
- Choose 'Virtual Devices' from the left-hand list
- Click 'New...'

- Choose a name for your device
- Set the *Target* to at least Android 2.2 – API Level 8
- Fill in an *SD Card* size.
- Enable the *Snapshot* option (this isn't essentially but significantly speeds up subsequent launches of the emulator!)
- Then click *Create AVD*

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

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

Configuring a real device

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

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

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

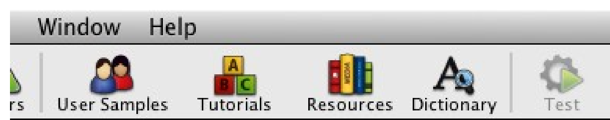
If you are running on Mac, there is no need to install any drivers as it 'just works'.

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

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

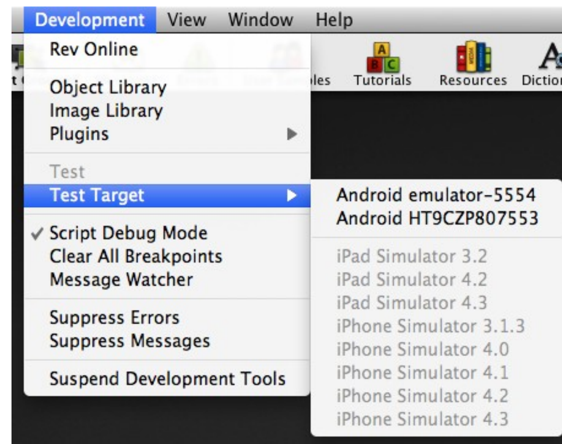
Testing an Android application

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



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

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



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

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

A First Project

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

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

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

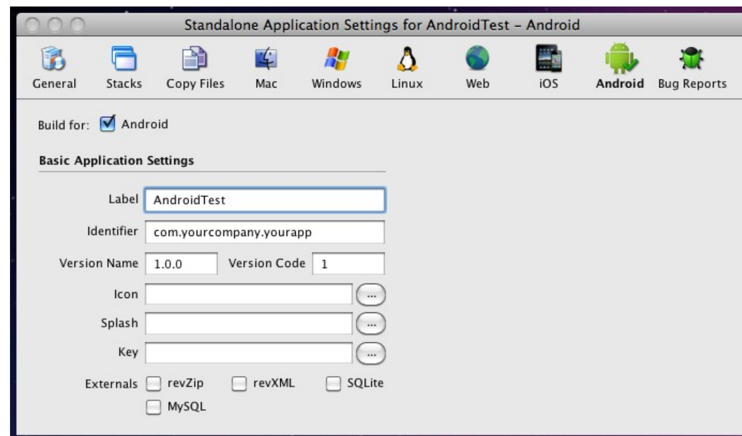
5. Save the *Hello World* stack.
6. Bring up the *Standalone Application Settings* dialog from the *File* menu, switch to the Android pane and make sure 'Build for Android' is checked.
7. Make sure your test stack is active and then click *Test* on the menubar.
8. Click the *Click Me* button in the simulator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

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

Configuring an Android Application

Setting manifest options

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



Here the numbered items are as follows:

1. The string to display as the label of the application on the Launcher screen.
2. The unique identifier to use for the application, using the standard reverse domain name convention.
3. A human readable version string for the application.
4. An integer indicating the version of the application – this is used by the OS to compare versions of packages as opposed to the human readable string.
5. The PNG image file to use as the icon on the Launcher.
6. The image file to use in the personal and educational splash screens (this is not used when building with a commercial license).
7. The key-store file to use to sign the application (this is only used when using 'Save as Standalone application').
8. The extensions to include in the application.

Adding a launcher icon

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

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

Adding a splash image (personal and educational)

If you are using a personal or educational license, then you are restricted in what can be displayed

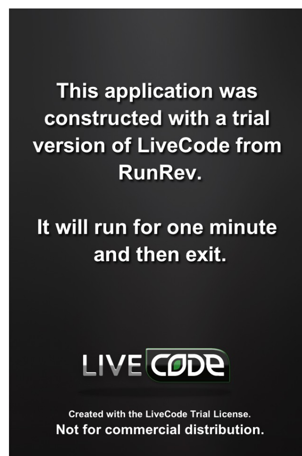
as the launch image. In this case you should provide a (square) PNG image that will be placed inside a LiveCode branded banner (see below).

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

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

Adding a default launch image (trial)

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



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

Deployment Features

Standalone builder messages

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

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

savingMobileStandalone *targetType*, *appBundle*

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

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

mobileStandaloneSaved *targetType*, *appBundle*

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

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

General Engine Features

Engine version

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

What doesn't work

The following features have no effect:

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

What does work

The following things do work as expected:

- rendering of controls with non-system themes (default is Motif theme)
- date and time handling
- gradients, graphic effects and blending

- any non-platform, non-system dependent syntax (maths functions, string processing functions, behaviors etc.)
- object snapshot commands
- revZip, revXML, dbSqlite and dbMysql

Windowing and Stacks

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

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

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

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

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

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

System Dialogs – answer and ask

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

The answer command can be used in this form:

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

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

The ask command can be used in this form:

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

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

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

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

Non-file URL access

The Android engine has support for fetching urls, posting to urls and downloading urls in the

background. Note that the iOS engine does not support libUrl, and as such there are some differences between url handling compared to the desktop.

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

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

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

To fetch the google home page you can do:

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

To post data to a website, you can use:

post tData **to url** tMyUrl

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

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

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

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

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

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

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

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

urlProgress *url,* "contacted"

urlProgress *url,* "requested"

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

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

urlProgress *url,* "downloaded"

urlProgress *url,* "uploaded"

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

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

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

libUrlDownloadToFile *url*, *filename*

Unlike the libUrl command of the same name, this command will block until the download is

complete, and will notify progress through the **urlProgress** message as described above.

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

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

Externals

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

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

Android Specific Engine Features

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

Limitations

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

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

Multi-touch events

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

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

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

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

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

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

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

Mouse events

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

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

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

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

Motion events

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

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

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

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

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

Hardware back key 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.

Accelerometer support

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

mobileEnableAccelerometer

mobileDisableAccelerometer

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

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

x,y,z,t

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

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

Photo album and camera support

Taking or choosing photos

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

mobilePickPhoto *source*

Here *source* is one of:

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

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

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

Keyboard Input

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

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

Configuring keyboard type

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

mobileSetKeyboardType *type*

Where *type* is one of:

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

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

If you wish to configure the keyboard options based on the field that is being focused, simply use

the commands in an *openField* handler of the given field. The keyboard is only shown after this handler returns, so it is the ideal time to configure it.

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

Orientation handling

The iOS engine includes support for automatic handling of changes in orientation and in so doing gains use of the smooth iOS standard animation rotation animation (note this replaces the previous approach of using *iphoneRotateInterface* which no longer does anything).

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.*

Resolution handling

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

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

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

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

Email composition

Basic support

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

The syntax of **revMail** is:

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

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

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

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

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

Advanced support

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cross-Mobile Note: *The compose mail feature works 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.*

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

Hardware and system version query support

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

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

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

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

Querying camera capabilities

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

mobileCameraFeatures([*camera*])

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

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

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

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

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

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

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

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

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)

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.

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.