iOS Deployment Release Notes (R17)

Table of Contents

Overview

LiveCode now incorporates facilities for deploying to iOS. These facilities include the ability to build iOS applications that run in a variety of simulator versions as well as on iPhone, iPod Touch and iPad devices.

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

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

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

Note: *iOS deployment is only supported on Macs running the latest versions of Leopard or Snow Leopard and require installation of an appropriate iOS SDK.*

Getting Started

Choosing an SDK

Before you can use iOS deployment, you need install the appropriate iOS SDKs available from Apple.

In order to get the iPhone SDK, you need to be 'registered iPhone developer'. You can register for this and download the SDK by visiting:

http://developer.apple.com/ios

LiveCode supports the following iOS SDKs:

Download	Platform	Simulators
Xcode 3.2.5 and iOS 4.2	Snow Leopard	4.2, 4.1, 4.0, 3.2
Xcode 3.2.4 and iOS 4.1	Snow Leopard	4.1, 4.0, 3.2
Xcode 3.2.1 and iOS 3.1.3	Snow Leopard	3.1.3
Xcode 3.1.4 and iOS 3.1.3	Leopard	3.1.3

Make sure you have at least one SDK installed, otherwise you will not be able to use the iOS deployment feature.

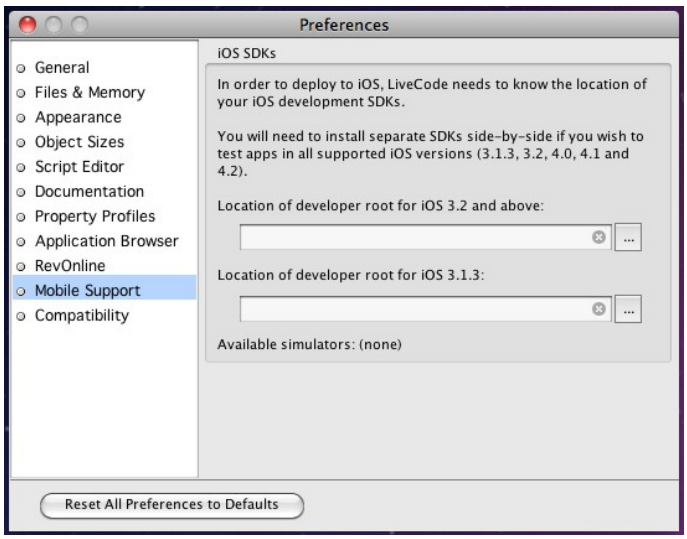
If you wish to test applications in all versions of the simulator (3.1.3, 3.2, 4.0, 4.1 and 4.2) then it is necessary to be running on Snow Leopard, and to have installed the iOS 4.2 SDK *and* the iOS 3.1.3 SDK in **separate** locations.

Note: *As a registered iOS developer you will be able to develop and run applications in the iPhone Simulator* ***only****. To build applications that can be run on an actual device you will need to enroll in the* iOS Developer Programme.

Configuring LiveCode

After you have installed an iOS SDK, it is necessary to tell LiveCode where to find it (or them, if you have installed more than one).

To configure the paths to your installed SDKs, use the *Mobile Support* panel in *Preferences*.



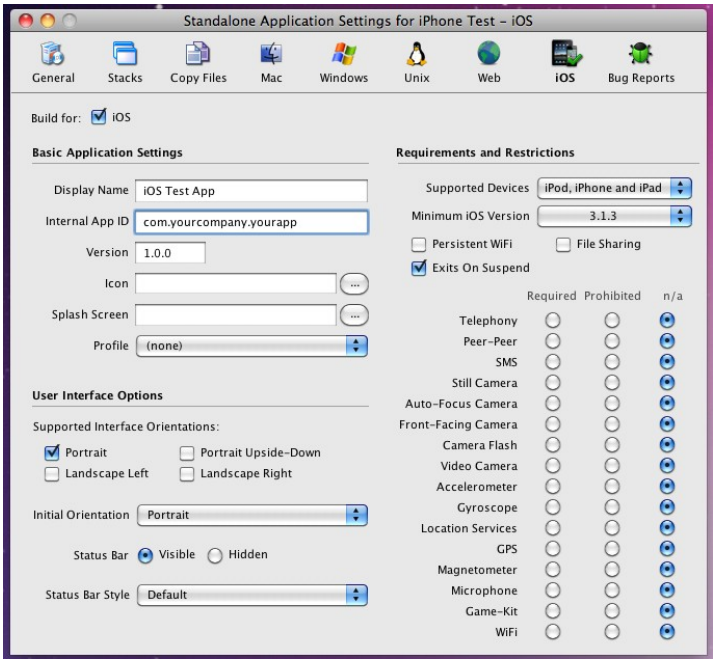
Use this pane to choose the correct SDK paths by using the '…' buttons next to the appropriate one. You should choose the folder you selected when installing the SDK. (This defaults to '/Developer' in the iOS SDK installers).

When you have successively chosen your SDK(s), the list of simulators that you will have available will be updated.

Note: *On startup if SDKs have not been previously configured, LiveCode will check to see if there is a recognised SDK at /Developer.*

Configuring an iOS standalone

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



This pane allows you to set the iOS-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 (bundle) name of your application on the *General* pane.

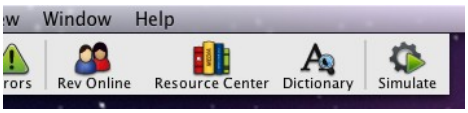
To make a stack build for iOS, simply check the *Build for iOS* button and configure any options that you wish.

Note: *Making a stack build for iOS disables building for any other platform, 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 iOS and desktop targets.*

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

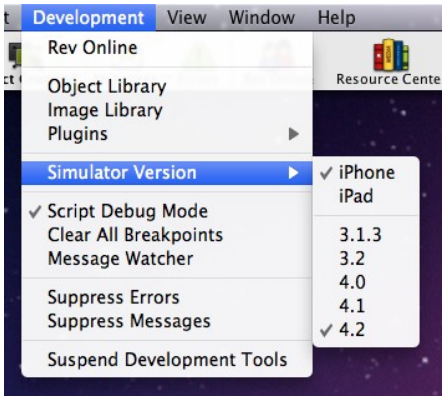
Running in the simulator

Once you have a stack configured for iOS, you can run it in the iOS Simulator by using the *Simulate* button on the menubar:



This button will be enabled for any stack that has been configured for iOS deployment, and clicking it will launch the stack in the simulator, terminating a running simulation if any.

You can configure which simulator version to use via the *Simulator Version* submenu of the *Developer* menu:



Here you can choose both version, and device type you wish to simulate. (Be aware that 3.2 is iPad only, and only 4.2 supports both iPad and iPhone/iPod Touch). Any setting you choose here will take effect the next time you use the *Simulate* button.

Note: *If the* Simulate *button remains disabled, even if you have configured a stack for iOS deployment, it probably means you haven't configured your SDKs correctly. In this case, check that there are available simulators in the* Mobile Support *pane of* Preferences.

A first project

Once you have installed an iOS 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 iOS pane and make sure 'Build for iOS' is checked.

7. Make sure you test stack is active and then click *Simulate* 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.

Building for a real device

Before you can begin testing your application on a real device, you will need to have several things in place:

1. Enrolment in the *iPhone Developer Programme*: this is required so that you can generate the necessary certificates and profiles.
2. A *iPhone Developer Certificate*: this is installed on your development machine and is used to digitally sign the application you wish to put onto an iPhoneOS device. Follow the instructions on the *Certificates* tab of the *iPhone Developer Program Portal*.
3. Registration of at least one iPhoneOS device in the program portal. You can add devices using the *Devices* tab of the *iPhone Developer Program Portal*.
4. An App ID for your application. You can create App IDs using the *App IDs* tab of the *iPhone Developer Program Portal*. *(Note that at this stage it isn't necessary for you to have a separate App ID for every app – you can use a single id for all your apps for testing/development purposes.)*
5. A provisioning profile tying together your test device's id, you app id and your certificate. These can be created using the *Provisioning* tab of *iPhone Developer Program Portal*.

Once you have all these things ready, you should find that the 'Profile' drop-down menu in the iOS pane of the *Standalone Settings* dialog is populated with any provisioning profiles you have installed.

With a suitable profile chosen, you can simply use the *Save as Standalone Application...* item in the *File* menu to build an iOS app bundle in the same was as you would build a standalone for any other platform.

The next thing to do is to install the bundle on your test device. To do this, start up Xcode, and choose **Window > Organizer**. This will bring an interface allowing you to manage the applications, devices and profiles you are using for development.

Next, make sure you have your test device connected to your machine and choose it from the left hand list. If you haven't used the device for development before, you will be prompted to do so, and you'll then be presented with a list of installed applications.

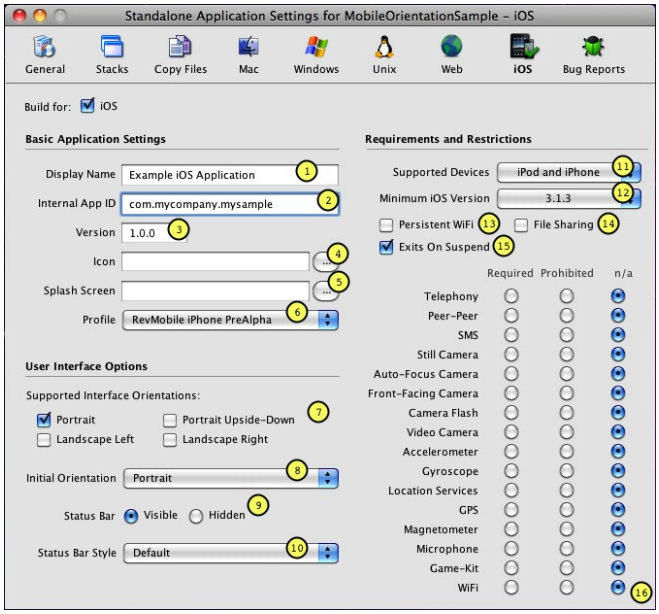
To get your newly prepared application on the device, simply drag the application bundle from the desktop into the *Applications* list – opting to install the appropriate provisioning profile if it has not been previously installed on the device.

Finally, navigate to the application on your device, and start it up!

Configuring an iOS Application

Setting plist options

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



Here the numbered items are as follows:

1. The string to display as the label of the application on the SpringBoard (CFBundleDisplayName).
2. The bundle identifier to use for the application, in conjunction with the App Id present in a provisioning profile, this uniquely identifies an application (CFBundleId).
3. The version of the application (CFBundleVersion).
4. The icon to display on the SpringBoard, see *Adding a SpringBoard icon* for more details (CFBundleIconFile and CFBundleIconFiles).
5. The image to use as the launch image (commercial), or the image to incoporate as the splash image (personal and educational), see *Adding a default launch image* or *Adding a splash image* for more details.
6. The provisioning profile to use when building the application to run on a device.
7. The set of (initial) interface orientations your application supports, iOS uses this key to determine what launch image to display (UISupportedInterfaceOrientations).
8. The initial orientation to start the application up in (UIInterfaceOrientation).
9. The initial visibility state of the status bar (UIStatusBarHidden).
10. The initial status bar style (UIStatusBarStyle).

11. The devices supported by the application, iOS uses this to determine if an application should launch on iPod/iPhones and whether it should run in iPod/iPhone emulation mode on iPads (UIDeviceFamily).
12. The minimum iOS version required by the application (MinimumOSVersion)
13. Determines whether the application requires a persistent WiFi connection (UIRequiresPersistentWiFi)
14. Determines whether the 'Shared Files' feature of iTunes is enabled for this application (UIFileSharingEnabled).
15. Determines whether the application quits when it is suspended on multi-tasking capable versions of iOS – for now leave this option enabled, the iOS engine does not yet fully support suspended operation (UIApplicationExitsOnSuspend).
16. These options determine what facilities the application requires or prohibits on the device in order to be launched (UIRequiredDeviceCapabilities).

More details of the plist options can be found in the iOS Reference Document.

Adding a SpringBoard icon

All applications currently installed on an iOS device are displayed on the *SpringBoard* – the home screen user interface you get presented with when the device is switched on.

Depending on what devices your application runs you should provide between 1 and 3 icons:

- <name>.png – a 57x57 icon for use on old (non-Retina) iPods and iPhones
- <name>-114.png – a 114x114 icon for use on Retina display capable iPods and iPhones
- <name>-72.png – a 72x72 icon for use on iPads

Here, <name> is anything you choose, the plugin will copy the files into the app bundle with the correct final name to be picked up by the OS.

You should always provide a 57x57 icon, and the plugin will automatically look for appropriately named icons at the other sizes to include.

Adding a default launch image (commercial)

On startup of an iOS application the SpringBoard will initially display a static image – this image stays on screen until the application has completely finished initializing and is ready to update the screen.

If you are using a commercial license then you have complete control over the launch image. You should provide between 1 and 4 images as follows:

- <name>.png – a 320x480 image for use on old (non-Retina) iPods and iPhones
- <name>@2x.png – a 640x960 image for use on Retina display capable iPods and iPhones
- <name>-Portrait.png – a 768x1024 image for use on iPads when in portrait mode
- <name>-Landscape.png – a 1024x768 image for use on iPads when in landscape mode

Here, <name> is anything you choose, the plugin will copy the files into the app bundle with the

correct final name to be picked up by the OS.

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).

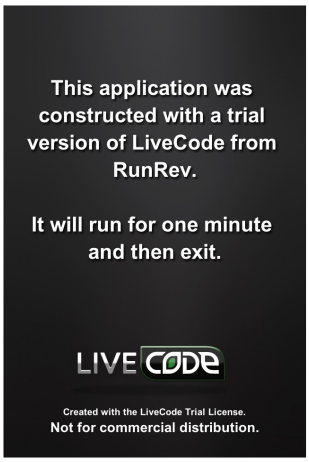
The plugin automatically generates a collection of launch images using this image depending on the target device settings you have specified in the plist.

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 iOS 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 iOS 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.

General Engine Features

Engine version

The current release of the iOS engine was derived from the 4.0 version of the desktop engine. This means that features present in 4.5.x that you might expect to work in the iOS engine will not be present at this time. In particular, the engine version is fixed at 4.0.0 and the build number at 950.

We are working on reintegrating the iOS port of the engine with the main desktop engine, and versioning will once again become unified in a future release.

What doesn't work

The following features have no effect:

- clipboard related syntax and functionality (planned for a future release)
- drag-drop related syntax and functionality (no support on mobile devices)
- printing syntax and functionality (planned for a future release)
- setting the mouseLoc (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)
- socket syntax and functionality (planned for a future release)
- audioclips/videoclips/player functionality (use the 'play' and 'play video' syntax described later)
- externals cannot be loaded (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.)

Debugging

At present the options available for debugging applications running on target devices is limited. Obviously, scripts will work in a similar fashion between Desktop and Mobile so this helps.

There is, however, a simple means of logging from an emulated target device. The LiveCode command form:

put *string*

Will write the string out to the standard error stream. These messages will be visible in *Console.app* when running in the simulator, and in the *Console* tab of the Xcode *Organizer* for a given target device while it is connected to the host computer.

Windowing and Stacks

The mobile 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.

System Dialogs – answer and ask

At present, only simple answer dialogs are supported in a restricted way. The answer command is mapped to the system standard *MessageBox* API call. The restricted syntax is:

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

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

Non-file URL access

The iOS 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.

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 “contacted”, pUrl

urlProgress “requested”, pUrl

urlProgress “loading”, pUrl, pBytesReceived, [pBytesTotal]

urlProgress “uploading”, pUrl, pBytesReceived, [pBytesTotal]

urlProgress “downloaded”, pUrl

urlProgress “uploaded”, pUrl

urlProgress “error”, pUrl, pErrorMessage

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.

iOS-specific engine features

This version of the LiveCode iOS engine includes a wide-range of features specific to iOS devices. These are described in the following sections.

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 iPhone'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.

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.

Motion events

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

- **motionStart** *motion*

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

Here *motion* is the type of motion detected by the device. As of iPhoneOS 3.0, 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).

Accelerometer support

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

iphoneEnableAccelerometer [*interval*]

iphoneDisableAccelerometer

Enabling the accelerometer will cause **accelerationChanged** events to be delivered to the current card of the defaultStack at the specified interval. The interval should be specified in seconds, and is the approximate time between delivery of messages. Note that the interval is constrained by hardware-specific minimums and maximums (which are left unspecified by Apple).

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.

Photo Picking Support

You can hook into the iPhone's native photo picker by using

iPhonePickPhoto *source*, [*maxwidth*, [*maxheight*]]

Here *source* is one of:

- *library* – the photo is taken from the device's photo library
- *camera* – the photo is taken using the device's camera
- *album* – the photo is taken from the device's recent camera roll

The *maxwidth* and *maxheight* parameters constrain the maximum size of an image. The chosen image will be scaled down proportionally to fit within the size specified. If either size specified is 0, then the parameter is ignored.

If the source type isn't availabe 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

Surprisingly, the SDK does not provide direct control over the iPhoneOS software keyboard. However, an attempt has been made to provide some level of support for text input entry. If you have a text field which is focusable (**traversalOn** true), then whenever it has focus the iPhone keyboard will appear and allow basic text editing functionality.

While it is possible to use the non-Roman keyboards to enter text, for scripts which have combining and/or input method type requirements the input will be incorrect. For example, languages such as Russian can be entered correctly, but Korean will not work as expected.

The auto-capitalization, auto-correction, copy/paste, undo/redo and selection point magnification features that are present in standard iPhone text entry fields are not supported.

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).

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.

You can fetch the current device orientation using the **iphoneDeviceOrientation()** 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 right
- *landscape right* – the device is being held upward with the home button on the left
- *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 **iphoneOrientation()** function. This returns one of *portrait, portrait upside down, landscape left* and *landscape right*. With the same meanings as for device orientation.

To configure which orientations your application supports use:

iphoneSetAllowedOrientations *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 **iphoneAllowedOrientations()** function.

To lock or unlock orientation changes for a time use:

iphoneLockOrientation and **iphoneUnlockOrientation**

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 **iphoneOrientationLocked()** function.

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 **iphoneDeviceOrientation()** 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.

Example: *You can find a simple stack using the orientation handling features in the IDE resources folder (open using the* ***Help > Example Stacks and Resources*** *menu item). The stack can be found at:* ***Mobile Examples/Orientation Example.livecode***

Resolution handling

The new iPhone 4 has a display with double the resolution in both horizontal and vertical directions. By default, iOS handles this by mapping one logical 'point' to two physical 'pixels' with applications (rev included) interpreting everything in terms of logical points. This means that apps targetted for older devices will run identically on the newer iPhone 4 devices.

As **the screenRect** and associated properties all deal in logical points, they do not reflect the actual device resolution at which the app is being displayed. To fetch the device screen's resolution in pixels use the **iphoneDeviceResolution()** function. This will return a string in the form *width*, *height* – with the values being given in pixels.

To use the full resolution of such high-resolution devices, use the command:

iphoneUseDeviceResolution *(true | false)*

If passed true, rev will ensure that co-ordinates and sizes specified in rev are treated as being in pixels, rather than logical points. In particular, when changed, a resizeStack message will be sent notifying in the size change of the current main-stack, and functions and properties (such as **the screenRect**) will reflect co-ordinates in pixels.

Note: The notion of pixel and logical point remains valid on older devices, its just that it is always 1-1 thus using this command will have no effect there.

Location handling

Basic support is present for CoreLocation – the framework that allows tracking of the device's position.

To start tracking the location of the device use:

iphoneStartTrackingLocation

To stop tracking the location of the device use:

iphoneStopTrackingLocation

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 ***locationError*** message is sent instead.

The current location of the device can be fetched by using the **iphoneCurrentLocation**() function. If location tracking has not been enabled this function returns empty. Otherwise 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.

Email composition

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:

- *not configured* – if the user has turned off or has not setup mail access on their device

- *cancel* – if the user chooses to cancel the send
- *saved* – if the user chose to save the message in drafts
- *sent* – if the user elected to send the email
- *failed* – if sending the email was attempted, but it failed

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.uracy will not be present.

File and folder handling

In general handling files and folders in the iPhone 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 iPhoneOS imposes strict controls over what you can and cannot access. Each application in iPhoneOS is stored in its own 'sandbox' folder (referred to as the *home* folder. An application is free to read and write files within this folder and its descendants, but is not allowed to access anything outside of this.

When an application is installed on a phone (or in the simulator) a number of initial folders are created for use by the application. You can locate the paths to these folders using the **specialFolderPath**() function with the following selectors:

- *home* – the (unique) folder containing the application bundle and its associated data and folders
- *documents* – the folder in which the application should store any document data (this folder is backed up by iTunes on sync)
- *cache* – the folder in which the application should store any transient data that needs to be preserved between launches (this folder is **not** backed up by iTunes on sync)
- *temporary* – the folder in which the application should store any temporary data that is not needed between launches (this folder is **not** backed up by iTunes on sync)
- *engine* – the folder containing the built standalone engine (i.e. the bundle). This is useful for constructing paths to resources that have been copied into the bundle at build time.

In general you should only create files within the documents, cache, and temporary folders. Indeed, be careful not to change or add any files within the application bundle. The application bundle is digitally signed when it is built, and any changes to it after this point will invalidate the signature and prevent it from launching.

Note: *Unlike (most) Mac OS X installs, the iPhoneOS filesystem is* ***case-sensitive*** *so take care to ensure that you consistently use the same casing for filenames when constructing them.*

System alert support

Support has been added for **the beepSound** and **beep** commands. These hook into iPhoneOS's standard *PlayPlayerSound* support.

To specify a sound to be played as the system sound, use **the beepSound** global property. This should be set to the filename of the sound to use when **beep** is executed. If you want no sound to play when using **beep**, simply set **the beepSound** to **empty**.

To perform a system alert, use the **beep** command. If no sound has been specified via **the beepSound** global property, the engine will request a vibration alert.

Note: *The iPhone has no default system alert sound so if a sound is required one must be specified by using* ***the beepSound****. The action of* ***beep*** *is controlled by the system and depends on the user's preference settings. In particular, a beep will only cause a vibration if the user has enabled that feature. Similarly, a beep will only cause a sound if the phone is not in silent mode.*

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 the iPhone (AVAudioPlayer, to be specific) and as such has support for a variety of formats including AIFF and 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 empty if there is no sound currently playing.

Important: *The iPhone simulator appears to have somewhat buggy support for sound playback via AVAudioPlayer – it will intermittently fail for no reason. Therefore, if you are using the* ***play sound*** *command be sure to test your application on a real device.*

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 iOS's built-in video playback support (MPMoviePlayer) and as such can use any video files supported by that, including mp4's.

On iPhoneOS 3.1.3, the video will always play with landscape orientation (there is no 'legal' way to change this). On iOS 3.2 and later, however, the orientation of the video will be tied to the current interface orientation.

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.

While a movie is playing, 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

Important: *The iPhone simulator appears to have somewhat buggy support for video playback via MPMoviePlayer in some versions. Therefore, if you are using the* ***play video*** *command be sure to test your application on a real device.*

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

Important: *Successfully launching a url will cause another application to open and the requesting application to be quit. The application will receive a* ***shutdown*** *message before this happens, however.*

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.

Note**:** *Strictly speaking the list returned by* ***fontStyles*** *isn't the font styles, but the font names and the list returned by* ***fontNames*** *isn't the font names but the font families.*

Visual effect support

The iOS engine now has support for a range of visual effects – including some specific to iOS. The following effects are available:

- scroll (up | left | down | right)
- reveal (up | left | down | right)
- push (up | left | down | right)
- dissolve
- curl (up | down)
- flip (left | right)

Speed can be controlled via the usual adjectives *very slow*, *slow*, *normal*, *fast* or *very fast*.

Status bar configuration support

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

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

iphoneShowStatusBar

iphoneHideStatusBar

To control the style of the status bar use the following command:

iphoneSetStatusBarStyle *style*

Where *style* is one of:

- *default* – the default mode for the device
- *translucent* – a semi-transparent status bar (in this case the stack will appear underneath it)
- *opaque* – a black status bar (in this case the stack will appear below it).

On iPad devices, anything other that *default* has no effect.

Locale and system language query support

You can query the list of preferred languages using the **iphonePreferredLanguages()** function. This returns a return-delimited list of standard language tags in order of user preference (for example “en”, “fr”, “de”, etc.)

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

Hardware and system version query support

You can now 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**. In the simulator this will return *i386* and on a real device this will return *ARM*.

To determine the type of device an application is running on use **the machine**. This will return one of:

- *iPod Touch* – the device is one of the iPod Touch models
- *iPhone* – the device is one of the iPhone models
- *iPhone Simulator* – the device is a simulated iPhone
- *iPad* – the device is the iPad
- *iPad Simulator* – the device is a simulator iPad

To determine the version of iPhoneOS the application is running on, use **the systemVersion**. For example, if the device has iPhoneOS 3.2 installed, this property will return *3.2*; if the device has iPhoneOS 3.1.3 installed, this property will return *3.1.3*.

Modal Pick-Wheel support

To cause a modal pickwheel to appear, use:

iphonePick *optionList*, *initialIndex*

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

A modal pick-wheel is also automatically displayed when the user touches an option menu control, functioning in much the same was as if a popup menu were to appear.

Note: *Although the iphonePick command does work on the iPad, it is not quite 'correct' when compared to Safari. A more consistent variant is being worked on.*

iOS Native Controls

Low-level support has been added for creating and manipulating some native iOS controls (views). There are generic set of commands and functions for creating and configuring certain UIView sub-classes which then layer above the currently displayed stack.

At present, there is an implementation for the UIWebView control (*browser*) and for the UIScrollView control (*scroller*).

To create a native control use:

iphoneControlCreate *controlType*

Where *controlType* is the type of control you wish to create – either “browser” or “scroller”. This command creates a new native control instance, and returns a unique (numeric) id for it in **the result**.

To destroy a native control use:

iphoneControlDestroy *id*

Where *id* is the numeric id returned by *iphoneControlCreate*.

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

iphoneControlSet *id*, *property*, *value*

Where

- *id* is the numeric id returned by *iphoneControlCreate*
- *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 **iphoneControlGet**(*id, property)*.

Control specific behavior can be invoked by using:

iphoneControlDo *id, 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 **iphoneControlTarget()** function to fetch the id 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.

Browser control (UIWebView)

A UIWebView control is created using a control type of “browser”. For full details of what the UIWebView control is capable of, and background about it see the iOS reference document.

Properties

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.
url	read/write	The url to be loaded into the web-view.
autoFit	read/write	Set to true or false to determine whether the page will be scaled to fit the rect of the control (wraps the *scalesPageToFit* property of UIWebView).
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).

Actions

iphoneControlDo *id,* "advance"

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

iphoneControlDo *id,* "retreat"

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

iphoneControlDo *id,* "reload"

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

iphoneControlDo *id,* "stop"

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

iphoneControlDo *id,* "load", *baseUrl, htmlText*

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

iphoneControlDo *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).

browserLoadRequest *url*, *type*

Sent when the given url has been requested. The reason for the request is specified in *type* which can be one of *click, submit*, *navigate*, *reload*, *resubmit* or *other.*

Not passing the message will cause the load request to not go ahead.

(This message is sent in response to the *webView:shouldStartLoadWithRequest:* delegate method).

browserLoadFailed *url*, *error*

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

Example: *You can find a simple stack using the native browser control features in the IDE resources folder (open using the* ***Help > Example Stacks and Resources*** *menu item). The stack can*

be found at: ***Mobile Examples/Browser Example.livecode***

Scroller control (UIScrollView)

A UIScrollView control is created using a control type of “scroller”. For full details of what the UIScrollView control is capable of, and background about it see the iOS reference document.

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 iOS 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

rect	read/write	The bounds of the control, relative to the top-left of the card. This is a comma-separated list of four integers, describing a rectangle.
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).
canBounce	read/write	Determines whether the scroller will 'bounce' when it hits the edge of the contentRect (maps to the UIScrollView *bounces* property). This is a boolean value.
canScrollToTop	read/write	Determines whether a touch on the status bar causes the scroll to scroll to the top (maps to the UIScrollView *scrollsToTop* property). This is a boolean value.
canCancelTouches	read/write	Determines whether the scroller is allowed to cancel an touch that has been passed through to the underlying controls when it thinks its a scroll gesture (maps to the UIScrollView *canCancelContentTouches* property). This is a boolean value.
delayTouches	read/write	Determines whether the scroller delays passing on touch-down events until it has determined whether it is the start of a scroll gesture

		or not (maps to the UIScrollView *delaysContentTouches* property). This is a boolean value.
pagingEnabled	read/write	Determines whether scrolling stops on multiples of the scroller's bounds (maps to the UIScrollView *pagingEnabled* property). This is a boolean value.
declerationRate	read/write	Determines the rate at which scrolling decelerates when a finger is lifted (maps to the UIScrollView *decelerationRate* property). This can be either *normal*, *fast* or a real number.
indicatorStyle	read/write	Determines the style of indicators to display (maps to the UIScrollView *indicatorStyle* property). This can be one of *default*, *white* or *black*.
indicatorInsets	read/write	Determines how far from the edge of the scrollers bounds, the indicators are inset (maps to the UIScrollView *scrollIndicatorInsets* property). This is a comma-separated list of four integers, describing the left, top, right and bottom inset distances.
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 (maps to the UIScrollView *showsHorizontalScrollIndicator* property). This is a boolean value.
vIndicator	read/write	Determines whether the vertical indicator should be displayed when scrolling (maps to the UIScrollView *showsVerticalScrollIndicator* property). This is a boolean value.
lockDirection	read/write	Determines whether scrolling is locked to the initial direction a drag occurs in (maps to the UIScrollView *directionalLockEnabled* property). This is a boolean value.
tracking	read-only	Returns true if the scroller is monitoring a touch for the start of a scroll action (maps to the UIScrollView *tracking* property). This is a boolean value.
dragging	read-only	Returns true if the scroller is currently performing a scroll action (maps to the UIScrollView *dragging* property). This is a boolean value.
decelerating	read-only	Returns true if the scroll is currently decelerating after a scroll action

(maps to the UIScrollView *decelerating* property).

This is a boolean value.

Actions

iphoneControlDo *id,* "flashScrollIndicators"

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

Messages

scrollerBeginDecelerate

This message is sent when scrolling is about to start decelerating.

scrollerEndDecelerate

This message is sent when scrolling has finished decelerating.

scrollerScrollToTop

This message is sent when the scroller is scrolled to top by touching the status bar.

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.

Example: *You can find a simple stack using the native scroller control features in the IDE resources folder (open using the* ***Help > Example Stacks and Resources*** *menu item). The stack can be found at:* ***Mobile Examples/Scroller Example.livecode***

Change Logs and History

Engine Change History

pre-alpha-3 (2010-03-04)	MW	Initial version.
pre-alpha-4 (2010-03-05)	MW	Bold and italic font styles now honoured in font selection Image picker no longer 'sticks' after selection GIF images now display Max width and height parameters added to iphonePickPhoto Import snapshot no longer crashes
pre-alpha-5 (2010-03-11)	MW	Unicode text will now display Umlauts and other non-ASCII characters will now display Return key now causes a newline in fields Crashes when changing image content have been fixed Export snapshot now makes images with the correct colors Rotating a non-masked images no longer causes corruption of the image
pre-alpha-6 (2010-03-18)	MW	Answer command now returns its the chosen button in 'it' Added support for detecting device orientation Added support for setting interface orientation Added basic support for CoreLocation Refined control hit-test for touch handling so disabled controls are not targetted. mouseLoc now reports the correct y-coordinate Added support for mail composition/sending Corrected file handling functions interpretation of '/' Added support for specialFolderPath() Fixed problem with incorrect display of animated GIFs
pre-alpha-7 (2010-03-29)	MW	Added basic support for 'play <soundfile>' Added support for 'beep' system alert Added support for 'launch url' Added support for 'the fontNames' and 'the fontStyles' Added support for 'uniEncode' and 'uniDecode' Added support for system date/time Fixed issue with engine not picking up 'Oblique' fonts for italic style Fixed issue with unicode text not displaying in fields on load Added support for 'engine' in 'specialFolderPath'
pre-alpha-8 (2010-04-12)	MW	Added support for targetting iPad Added support for 'the systemVersion' Added support for 'the machine' Added support for 'the processor' Fixed problem with orientation returning portrait mispelt Improved responsiveness of image picker Added support for iphonePickPhoto on the iPad
pre-alpha-10 (2010-08-12)	MW	Added support for 'play video <filename/url>' Fixed issues with support for environment properties (the

		systemVersion, the processor, etc.)
		Added support for 'the sound'
		Fixed issue with garbage being returned from specialFolderPath in some cases.
		Added support for 'libUrlDownloadToFile'
pre-release-14 (2010-11-10)	MW	Added support for 'load url'
		Added support for 'post url'
		Added support for status bar configuration
		Added support for building for appstore/ad-hoc
		Added support for visual effects
		Fixed issue with iphonePickPhoto crashing on iOS4
		Fixed issue with some PNGs not displaying correctly
		Fixed issue with graphic effects have inverted colors
		Fixed issue with black screen appearing on startup
		Fixed issues with landscape orientation mode
pre-release-17 (2010-12-01)	MW	Added support for browser native control
		Added support for scroller native control
		Added support for querying current locale and preferred languages
		Added support for 'movieTouched' message while movie playing
		Added supoprt for 'play stop' command while movie playing
		Added support for building iOS apps with evaluation licenses
		Added support for modal pickwheel, and hooked into option menus
		Changed support for orientation handling to leverage built-in iOS mechanism
		Fixed various glitches with movie playback
		Fixed issue with entering accented characters with the iOS keyboard
		Fixed issue with visual effects not working correctly in non-portrait orientation
		Fixed issue with 'the mouseColor' causing a crash

iOS Deployment Change History

pre-alpha-3 (2010-03-04)	MW	Initial version.
pre-alpha-4 (2010-03-05)	MW	Bundle identifier setting no longer lost on reload
pre-alpha-5 (2010-03-11)	MW	Project settings are no longer lost when adding/removing files
pre-alpha-10 (2010-08-12)	MW	Added support for configuring SDK roots
		Added support for adding folders of files to the app bundle.
pre-release-14 (2010-11-10)	MW	Added support for ad-hoc and store profiles
		Added support for specifying a splash screen
		Added support for copying in icons of different resolutions
		Added support for plist configuration
		Fixed issue with app bundle not being deleted before rebuilding
release-17 (2010-12-01)	MW	Integrated plugin's functionality into IDE
		Simulate start/stop buttons replaced by single menubar 'Simulate' button
		Deploy button replaced by standard 'Save as Standalone

Application' action
Plist editor integrated as new Standalone Builder pane
Simulator selection moved to Simulator Version menu item of Development menu
SDK configuration moved to 'Mobile Support' pane of preferences
Added support UIFileSharingEnabled plist tag
Added ability to choose device type for simulator (iPad/iPhone)
Fixed issue with launch image filenames not being correctly computed (and thus failing to copy into the bundle)

Document History

Revision 1 (2010-03-04)	MW	Initial version.
Revision 2 (2010-03-05)	MW	Added documentation for new iphonePhotoPick parameters
Revision 3 (2010-03-11)	MW	Improved consistency of syntax specifications and use Refined documentation for touch events Added new section about mouse events Added new section on restrictions to dynamic features Restructured headings to make sure PDF index works
Revision 4 (2010-03-18)	MW	Added section on orientation handling Added section on location handling Refined statements about the mouseLoc Refined description of touch handling with regard to hit-testing Clarified support for dynamic chunks Added section on email composition Added section file handling Clarified blocking behavior of non-file url's
Revision 5 (2010-03-29)	MW	Added section on system alerts Added section on sound support Added section on url launching Added section on font querying Added description of *engine* parameter to specialFolderPath
Revision 6 (2010-04-12)	MW	Revised setting up your system with regard iPad support Added section on hardware and system version querying
Revision 7 (2010-08-12)	MW	Revising initial sections to include details of SDK configuration and requirements. Revised 'Sound file support' section to include 'the sound'. Added 'Video file support' section. Revised 'Non-file url' section to include 'libUrlDownloadToFile' Revised 'The revMobile Plugin' section to include changes to UI. Revised 'Projects and Files' section to include details about adding folders.
Revision 8 (2010-09-16)	MW	Rebranded from revMobile to iOS Deployment Rebranded from rev* to LiveCode Removed section on dynamic language features as no longer relevant.
Revision 9 (2010-11-10)	MW	Various edits to improve language. Expanded section on url commands

		Added section on visual effects Added section on status bar configuration Revised 'The Deployment Plugin' section Revised the non-test deployment section Added a section on the plist editor Added a section on launch images Added a section on splash images
Revision 10 (2010-12-01)	MW	Rewrote and reorganised initial sections to reflect new integration into the IDE. Rewrote section on orientation handling. Added section on native controls and further sub-sections on browser and scroller controls. Added section of locale and system language query support. Revised the play video section.
Revision 11 (2010-12-04)	MW	Added section on 'Engine Version'