Mobile Specific Engine Features

The LiveCode Android and iOS engines includes a wide-range of features specific to their specific 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 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 interprets 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 respond in a similar way as they do in the desktop version. You receive the standard mouse events and the *mouseLoc* is kept updated appropriately.

Motion Events

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

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

Here *motion* is the type of motion detected by the device.

When the motion starts, the current card of the defaultStack receives **motionStart** and when the motion ends it received **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).

Sensor Tracking

Four different sensor can be tracked:

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

The names detailed in bold are used to reference the sensors.

Sensor Availability

mobileSensorAvailable *(sensor)*

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

Start Tracking Sensor

mobileStartTrackingSensor *sensor, [loosely]*

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

The parameter loosely is a boolean determining how detailed the readings from the sensors should be.

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

Stop Tracking Sensor

mobileStopTrackingSensor *sensor*

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

Sensor Update Messages

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

locationChanged *latitude, longitude, altitude*

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

headingChanged *heading*
heading – the heading of the device, in degrees relative to true north if available, otherwise relative to magnetic north
accelerationChanged *x, y, z*

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

rotationRateChanged *x, y, z*

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

If at any point there is an error tracking one of the sensors, the trackingError message is sent:
trackingError *sensor, errorMessage*

Getting a Sensor Reading

In addition to the update messages that are sent, you can get the reading of any sensor you are tracking using the function
mobileGetSensorReading
mobileSensorReading *(sensor, [detailed])*
The boolean parameter detailed determines the amount of detail present in the data returned. If this is false, the data returned is a comma separated list. If true, an array is returned. By default, *detailed* is false.
The data returned depends upon the sensor.
Location – a comma separated list of the latitude, longitude and altitude of the device. If detailed is true an array containing the keys latitude, longitude, altitude, time stamp, horizontal accuracy, vertical accuracy, speed and course is returned.
If the latitude and longitude could not be measured, those values together with the horizontal accuracy key will not be present. If the

altitude could not be measured, that value together with the vertical accuracy will not be present.
Heading – the heading of the device in degrees. If detailed is true an array containing the keys heading, magnetic heading, true heading, time stamp, x, y, z and accuracy is returned.
Acceleration – a comma separated list of the acceleration in the x, y and z axes. If detailed is true an array containing the keys x, y, z and timestamp is returned.
Rotation Rate – a comma separated list of the rate of rotation around the x, y and z axes. If detailed is true an array containing the keys x, y, z and timestamp is returned.

Email Composition

Basic Support

A version of **revMail** has been implemented that hooks into the mobile messaging 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 on iOS, the result is 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: Once you have 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 the mobile 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 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"]
mobileComposeMail 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
Note: There are hard limits imposed by iOS of the size of attachments that can be made. This is not precisely specified anywhere but appears to be around 16Mb based on forum threads.
Upon completion of a compose request the result is *unknown* on Android devices. On iOS devices the result is set to one of the following:

- *sent* – the email was sent successfully
- *failed* – the email failed to send
- *saved* – the email was not sent, but the user elected to save it for later
- *cancel* – the email was not sent, and the user elected not to save it for later
- *not configured* – the device is not configured to send email

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

Text Messaging Support

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

mobileComposeTextMessage recipients, [body]

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

Note: Once you have called the **mobileComposeTextMessage** command you have no more control over what the user does with the message – they are free to modify it and the addresses as they see fit.

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

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

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

File and Folder Handling

In general handling files and folders in the mobile engine is the same as that on the desktop. All the usual syntax associated with such operations 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 iOS and Android impose strict controls over what you can and cannot access.

File and Folder Handling on Android

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.

File and Folder Handling on iOS

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)
- *library* – the folder in which the application can store data of various types. In particular, data private to the application should be stored in a folder named with the app's bundle identifier inside library. (this folder is 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. Also note that the Simulator has the same case-sensitivity as the host system and not the device.

System Alert Support

To perform a system alert, use the **beep** command.
On iOS, this hooks into the 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**, 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 requests 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 only causes a vibration if the user has enabled that feature. Similarly, a beep only causes a sound if the phone is not in silent mode.

Vibration Support

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

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 the device and as such has support for a variety of formats.
You can monitor the current sound being played by using **the sound** global property. This either returns the filename of the sound currently being played, or “done” if there is no sound currently playing.

Multi-channel sound support

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

Playing Sounds

To play a sound on a given channel use the following command:
mobilePlaySoundOnChannel *sound, channel, type*
Where sound is the sound file you wish to play, channel is the name of the channel to play it on and type is one of:

- *now* – play the sound immediately, replacing any current sound (and queued sound) on the channel.

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

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

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

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

soundFinishedOnChannel *channel, sound*

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

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

mobileStopPlayingOnChannel *channel*

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

mobilePausePlayingOnChannel *channel*

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

mobileResumePlayingOnChannel *channel*

Channel Properties

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

iphoneSetSoundChannelVolume *channel, volume*
iphoneSoundChannelVolume(*channel* **)**
Here channel is the channel to affect, and volume is an integer between 0 and 100 where 0 is no volume, 100 is full volume. Changing the volume affects the currently playing sound and any sounds played subsequently on that channel.
Note: You can set the volume of a nonexistent channel and this results in it being created. This allows you to set the volume before any sounds are played. If you attempt to get the volume of a non-existent channel, however, empty is retuned.
To find out what sounds (if any) are currently playing and are scheduled for playing next on a given channel use:
mobileSoundOnChannel(*channel* **)**
mobileNextSoundOnChannel(*channel* **)**
These return empty if no sound is currently (scheduled for) playing (or the channel does not exist).
To query a channel's current status use **mobileSoundChannelStatus()**. This returns one of the following:

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

Managing Channels

To get a list of the sound channels that currently exist use:
mobileSoundChannels()
This returns a return-delimited list of the channel names.
Sound channels persist after any sounds have finished playing on them, retaining the last set volume setting. To remove a channel from memory completely use:
mobileDeleteSoundChannel *channel*

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

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 (video-file | video-url)*
The video is played fullscreen, and the command does not return until it is complete, or the user dismisses it.
If a path is specified it is 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 is streamed.
The playback uses the built-in video playback support and as such can use any video files supported by the device.
Note: On iPhoneOS 3.1.3, the video is always played with landscape orientation (there is no 'legal' way to change this). On iOS 3.2 and later, the orientation of the video is 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.
When a movie is played without controller, any touch on the screen results in a **movieTouched** message being sent to the object's whose script started the video. The principal purpose of this message is allow the**play stop** command to be used to stop the movie. e.g.

```
on movieTouched
play stop
end movieTouched
```

Note: The **movieTouched** message is not sent if the video is played with **showController** set to true.
Playing a video can be made to loop by setting **the looping of the templatePlayer** to true before executing the play video command.
Note: Looping video is not supported below iOS 3.2.

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

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 returns 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.

Visual Effect Support for iOS

The iOS engine supports 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.

For the *flip* visual effect, the background behind the flip will be taken from the background color of
the current stack – i.e. the card is cut out and flipped over the stack.

Status Bar Configuration Support for iOS

You can 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.

Runtime Environment Querying for iOS

You can fetch numerous pieces of information about the environment in which the current application is running with the following syntax.

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.
You can fetch the current device's unique system identifier with the **iphoneSystemIdentifier()** function. This returns a string in the standard UUID/GUID format.
The bundle identifier for the current application can be queried with **iphoneApplicationIdentifier()** function. This returns the identifier specified in the iOS standalone settings, and is useful (among other things) for creating a private folder in specialFolderPath("library") following Apple guidelines.

Managing Redraws of iOS

The function **iphoneSetRedrawInterval** can be used to manage the way LiveCode handles redraws. By default, LiveCode updates the screen immediately after any command that requires it. This means that several small screen updates may occur in quick succession when animation is combined with other dynamic screen elements. On mobile devices this can affect smoothness of animation where it would be better if multiple frequent redraws were replaced with a single periodic redraw. Setting the**iphoneSetRedrawInterval** enables this behavior, where the screen is updated at a fixed interval tied to iOS's redraw rate.
iphoneSetRedrawInterval *frameInterval*
frameInterval – A number specifying how often LiveCode should update the screen in line with the screen's refresh rate.
0 – Turn off synchronized redraws and revert to default LiveCode redrawing behavior.
1 – Redraw every time iOS redraws.
2 – Redraw on every other iOS redraw.
x – Redraw every x iOS redraws.

Activity Indicator for iOS

iOS provides a native animated activity indicator that sits above all other components and is used to indicate that an app is busy processing.

Use the **iphoneActivityIndicatorStart** command to display a native iOS activity indicator on the top of the LiveCode stack that is running.

iphoneActivityIndicatorStart *[type], [xposition, yposition]*

Here, type can be one of:

- *gray* – default, displays a small gray spoked animation
- *white* – displays a small white spoked animation
- *whiteLarge* – displays a large white spoked animation

The xposition and yposition specify the location in pixels of the activity indicator. If a location is not specified, then the animation is positioned in the middle of the screen.

You can turn the activity indicator off by calling:

iphoneActivityIndicatorStop

Locale and System Language Query Support

You can query the list of preferred languages using the **mobilePreferredLanguages()** 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 **mobileCurrentLocale()** function. This returns a standard locale tag (for example “en_GB”, “en_US”, “fr_FR”, etc.)

Modal Pick-Wheel Support

You can present the user with a list of choices to pick from using standard Android and iOS interface elements using:
mobilePick *optionList, initialIndex, [style],* ***My button*** (for Android)
mobilePick *optionList, initialIndex, [optionList, initialIndex, …], [style],* ***My button*** *, [view]* (for iOS)
Where optionList is a return-delimited list to choose from, and initialIndex is the (1-based) index of the item to be initially highlighted. The item the user chooses is returned in **the result**.

Modal Pick-Wheel Support on Android

The initialIndex is the (1-based) index of the item to be initially highlighted.
The optional style parameter determines the type of display used If equal to “checkmark” a check- mark (radio button) will be put against the currently selected item.
The optional button parameter specifies if “Cancel” and/or “Done” buttons should be forced to be displayed with the picker dialog.

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

If the ‘Cancel’ button is displayed, then any selection made by the user can be canceled, the result contains the initial index.
If the ‘Done’ button is displayed, then the result contains the initial index.
Pressing the back key has the same result as the ‘Cancel’ button.

Modal Pick-Wheel Support for iOS

A pick-wheel with multiple columns can be created by specifying more than one optionList initalIndex pair. For multi-column pick-wheels, the result will be a comma separated list of the chosen items, on item for each column.

On the iPhone, a standard Action Sheet pops up containing the standard pick-wheel user interface element; and on the iPad, a standard pop-over is presented with a list to choose from.

There are two modes of operation of the pick command, depending on the value of initialIndex.

If *initialIndex* is non-zero, the operation will act as a means to change an existing selection. The item specified by the initial index will be hilited (checked or hilited on iPad), and will be returned by default in the case the user does not choose a new item.

If *initialIndex* is zero, the operation will act as a means to select from a list of options. The user will be able to cancel the operation by either clicking 'Cancel' (iPhone) or touching outside of the pop-over (iPad). If the operation is cancelled, 0 will be returned; otherwise the selected item will be returned.

The optional style parameter determines the type of display used on the iPad. If equal to "checkmark" a check-mark (tick) will be put against the currently selected item. If not present, the currently selected item will be hilited with the (standard) blue background.

The optional button parameter specifies if "Cancel" and/or "Done" buttons should be forced to be displayed with the picker dialog. The default behavior is device dependent, exhibiting the most native operation.

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

The optional view parameter specifies the type of view to be displayed when showing a single column of date on an iPad. By default a standard pop-over is displayed with a single column of data. If “picker” is specified, then the single column of data is replaced with a single picker wheel.

Date Picker Support

You can present the user with a standard iOS date picker using:
mobilePickDate *[mode], [initial], [min], [max], [step],* ***My button***
The display style of the date picker will be determined by the users current calendar style as
configured in Settings.
The mode parameter determines the mode of the date picker and can be one of the following:

- date
- time
- dateTime (iOS only)

The mode defaults to date.
The initial parameter determines the initial date to be displayed by the date picker. If this is empty, the current date will be used. This should be a time in seconds since the Unix Epoch.
The min parameter is the start range of the date picker. If this value is empty, there is no lower boundary. The value is ignored if min is greater than max. This should be a time in seconds since the Unix Epoch.
The max parameter is the end range of the date picker. If this value is empty, there is no upper boundary. The value is ignored if max is less than min. This should be a time in seconds since the Unix Epoch.
The step parameter in iOS only specifies the minute interval size. This parameter is ignored if mode is set to “date”. The default is 1.

The optional button parameter in iOS only specifies if “Cancel” and/or “Done” buttons should be forced to be displayed with the picker dialog. The default behavior is device dependent, exhibiting the most native operation.

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

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

Media Picker Support

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

mobilePickMedia *[multiple], [type…]*

The optional parameters are only supported by iOS.

Set multiple to true if you want to allow the user to pick more than one item.

You can specify the type of media item the user is to select from by passing one or more of the
following:

- *music* – specifies that the user should be allowed to select music items
- *podCast* – specifies that the user should be allowed to select pod casts
- *audioBook* – specifies that the user should be allowed to select audio books

If no types are passed, all media items will be displayed.

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

Contact Access

Interaction with the contact list can be controlled either via native user interfaces or directly from the LiveCode syntax.

UI Contact Access Features

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

Creating a Contact

You launch the native mobile contact creation dialog by calling the command:

mobileCreateContact

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

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

Picking a Contact

The user can select a contact from the contact list by using the command:

mobilePickContact

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

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

Showing a Contact

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

mobileShowContact *contactID*

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

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

Updating a Contact

A contact can be pre-populated with information before launching an iOS contact creation dialog by 45 using the command:
mobileUpdateContact *contactArray, [title], [message], [alternateName]*
This feature allows interaction with the contact creation process to be streamlined for the user. If an application is already aware of some of the contact details that the user has to complete, then that data can be entered into the new contact automatically.
The information to pre-populate contact information is provided in form of an array. The array structure is detailed under heading "Contact Array Structure".
The optional parameters, only supported by iOS, are as follows:

- *title* – the header shown in the contact UI
- *message* – any message to be added to the contact UI
- *alternateName* – an alternate name to be displayed in the contact UI

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

Syntax Contact Access Features

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

Contact Array Structure

The handlers **mobileUpdateContact**, **mobileAddContact** and **mobileGetContactData** all use a common contact array format. The structure of the array is defined as follows:

Person Information

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

- *firstname* – the first name
- *middlename* – the middle name
- *lastname* – the last name
- *alternatename* – the alternative name
- *nickname* – the nick name

- *firstnamephonetic* – the phonetic transcription of the first name
- *middlenamephonetic* – the phonetic transcription of the middle name
- *lastnamephonetic* – the phonetic transcription of the last name
- *prefix* – the name prefix
- *suffix* – the name suffix
- *organization* – the name of the organization
- *jobtitle* – the job title
- *department* – the name of the department
- *message* – a person message
- *note* – a person note

E-Mail Addresses

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

- *home* – the home e-mail address
- *work* – the work e-mail address
- *other* – an alternative e-mail address

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

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

tContactData["email"]["home"][1]

Telephone Numbers

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

- *mobile* – the mobile telephone number
- *iphone* – the iPhone telephone number (iOS only)
- *main* – the main telephone number
- *home* – the home telephone number
- *work* – the work telephone number
- *homefax* – the home FAX number

- *workfax* – the work FAX number
- *otherfax* – an alternative FAX number (iOS 5.0 and later)
- *pager* – the pager number
- *other* – an alternative telephone number

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

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

tContactData["phone"]["mobile"][1]

Addresses

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

- *home* – the home address
- *work* – the work address
- *other* – an alternative address

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

Each individual address has the following keys:

- *street* – the address's street
- *city* – the address's city
- *state* – the address's state
- *zip* – the address's ZIP code
- *country* – the address's country
- *countrycode* – the address's country code

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

tContactData["address"]["home"][1]["street"]

Adding a Contact

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

mobileAddContact *contactArray*
The contact array structure is detailed under heading “Contact Array Structure”.
The result of this command returns either “empty” if no contact was created or the ID of a successfully created contact.

Finding a Contact

The contact list database can be queried, based on the contact name, using the command:
mobileFindContact contactName
It is possible to provide parts of the contact’s name that is to be queried as the argument *contactName*. The first letter of the given name or surname would be sufficient to provide a search for a given contact.
The result of this command returns either “empty” if no contact could be found or a comma delimited list of IDs of the contacts that match the search.

Removing a Contact

A contact can be removed from the contact list by using the command:
mobileRemoveContact *contactID*
The result of this command returns either “empty” if no contact with the provided ID could be found or the ID of the contact that was deleted.

Getting Contact Data

Information stored against a particular contact can be retrieved by calling the function:
mobileGetContactData *(contactID)*
This function extracts all of the contact fields that are supported in LiveCode and returns them in form of an array with the array keys representing the tokens and the corresponding array values representing the contact specific information.
The contact array structure is detailed under heading “Contact Array Structure”.

Idle Timer Configuration

By default, Android and iOS dim the screen and eventually lock the device after periods of no user-interaction.
To control this behavior, use the following commands:
mobileLockIdleTimer
mobileUnlockIdleTimer
Locking the idle timer increments an internal lock count, while unlocking the idle timer decrements the lock count. When the lock count goes from 0 to 1, the idleTimer is turned off; when the lock count goes from 1 to 0, the idleTimer is turned on.
To determine whether the idleTimer is currently locked (i.e. turned off) use **mobileIdleTimerLocked()**.

Querying Camera Capabilities

To find out the capabilities of the current devices camera(s), use the following function:
mobileCameraFeatures ([*camera*])
The camera parameter is a string containing either “rear” or “front”. In this case, the capabilities of that camera are returned. These take the form of a comma-delimited list of one or more of the following:

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

If the returned string is empty it means the device does not have that type of camera.
If no camera parameter is specified (or is empty), then a comma-delimited list of one or more of the following is returned:

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

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

If the returned string is empty it means the device has no cameras.
Note: At this time, Android can only detect whether there are front and/or back cameras and whether they can take photos.

Clearing Pending Interactions

As interaction events (touch and mouse messages) are queued, it is possible for such messages to accumulate when they aren't needed. In particular, when executing 'waits', 'moves' or during card transitions.
To handle this case, the **mobileClearTouches** command has been added. At the point of calling, this will collect all pending touch interactions and remove them from the event queue.
Note: This also cancels any existing mouse or touch sequences, meaning that you (nor the engine) will not receive a mouseUp, mouseRelease, touchEnd or touchCancel message for any current interactions.
A good example of when this command might be useful is when playing an instructional sound:

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

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

Busy Indicator

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

mobileBusyIndicatorStart *style, [label], [opacity]*

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

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

The optional opacity parameter is supported by iOS only and is used to determine the opacity of the busy indicator’s background.This should be an integer between 0 and 100. If no value is passed, this defaults to 42. To dismiss the dialog, use the mobileBusyIndicatorStop command.

mobileBusyIndicatorStop

Local notifications

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

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

Use the command mobileCreateLocalNotification to schedule a notification with the Android or iOS.

- *alertBody* – the text that is to be displayed on the notification dialog, that is raised when the application is not running

- *alertButtonMessage* – the button text on the notification dialog, that is to appear on the button that launches the application, when the application is not running (iOS only)
- *alertPayload* – a text payload that can be sent with the notification request. This payload is presented to the user via the localNotificationReceived message (iOS only)
- *alertTime* – the time at which the alert is to be sent to the application
- *playSound* – boolean to indicate if a sound is to be played when the alert is received
- *badgeValue* – the number value to which the badge of the application logo is to be set. 0 hides the badge. Greater than 0 displays the value on the badge.

A return delimited list of all the currently pending notifications can be fetched using the function:
mobileGetRegisteredNotifications()
You can query a given notification using the function:
mobileGetNotificationDetails(*notification* **)**
This returns an array with the following entries:

- *body* – the text that is to be displayed on the notification dialog (iOS) or status bar entry (Android) when the application is not running
- *title* – the button text on the notification dialog (iOS) or the title of the status bar entry (Android)
- *payload* – the text presented to the app via the localNotificationReceived message
- *play sound* – boolean indicating if a sound is to be played when the notification is received
- *badge value* – the number value which should be displayed on the app logo (iOS) or on the status bar icon (Android) when the notification is received. No number will be displayed if this is zero

To cancel a notification use command:
mobileCancelLocalNotification *notification*

Here, the *notification* parameter is a value returned by **mobileGetRegisteredNotifications()**.
To cancel all pending notifications, use the command **mobileCancelAllLocalNotifications**.
When you app receives a notification, the message **localNotificationReceived** is sent.
localNotificationReceived *message*
Here, the message parameter is the payload specified when the notification was created.

Push Notifications

Push notifications allow apps to avoid frequently polling for the availability of new remote data by providing a mechanism whereby notifications can be sent to the mobile device.
The following lessons provide detailed information on how to use Push Notifications:
How do I use Push Notifications with iOS
How do I use Push Notifications with Android

Custom URL Schemes

Specifying a custom URL allows you app to be woken up when the given URL is invoked.
To specify a custom URL, add the desired URL to the “URL Name” field of the standalone application builder. For example, if you specify “myURL” as the URL name, then when the URL myURL:// is invoked, if installed, your app will be woken.
Extra parameters can be passed in the URL in the following format:
myURL://
myURL://some/path/here
myURL://?foo=1&bar=2

myURL://some/path/here?foo=1&bar=2
If you app is woken by a custom URL, the message **urlWakeUp** is sent to the current card.
urlWakeUp *urlString*
A single parameter will be passed detailing the URL used to launch your app. This value can bet retrieved at any point using the function mobileGetLaunchURL(). If the app was not launched from a URL then this will return empty.

In App Advertising

Ads are supplied by our ad partner inneractive and come in three different types: banner, full screen and text. Before you can begin placing ads, you must first register your app with inneractive.
To do this, sign up with inneractive at the following URL: http://runrev.com/store/account/inneractive/. Once successfully signed up with inneractive you must generate a key for your app. Do this by clicking on the “Add App” tab of the inneractive dashboard and following the instructions provided.
Once you have a key for your app, you must register this with LiveCode using the **mobileAdRegister**command. You are now ready to place ads, using the **mobileAdCreate** command.

Registering Your App Key

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

Creating and Managing Ads

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

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

The parameters are as follows:

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

Ads can be deleted at any time using to command:

mobileAdDelete *ad*

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

mobileAdGetTopLeft *(ad)*

mobileAdSetTopLeft *(ad, topLeft)*
The top left is the pixel coordinates of the top left corner of the ad.
You can get and set the visibility of an ad using the following:
mobileAdGetTopVisible *(ad)*
mobileAdSetTopVisible *(ad, visible)*
The visible is a boolean, set to true if the ad is visible, false otherwise.
A list of all the currently active ads can be fetched using the function:
mobileAds()
This returns a return-delimited list of the ad names.

Messages

When an add is loaded or refreshed, the message adLoaded will be sent to the current card:
adLoaded *default*
Here, default is a boolean, set to true if the loaded ad is a default ad.
If a user clicks on an ad, the adClicked message will be sent to the current card.
adClicked
If an ad fails to load the adLoadFailed message will be sent to the current card:
adLoadFailed
If an ad is about to resize the adResizeStart message will be sent to the current card:
adResizeStart
When an ad has finished resizing the adResizeEnd message will be sent to the current card:
adResizeEnd
If an ad is about to expand the adExpandStart message will be sent to the current card:
adExpandStart

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

In App Purchasing

iOS Setup

In order to perform in app purchases you must first configure a number of items in iTunesConnect.

Setup a Contract

The first thing to do is make sure you have setup an ‘iOS Paid Applications’ contract. This is done in the ‘Contracts, Tax and Banking’ section of the iTunesConnect account.

Setup the In-App Purchase

If you have not already added your app to iTunesConnect you have to do that first.
http://lessons.runrev.com/s/lessons/m/4069/l/33065
Create your in-app purchase by selecting the application you want to associate it to, and then from the apps mini-page clicking ‘Manage In-App Purchases’ and then ‘Create New’. You will be asked to select a purchase type, set the price and provide various pieces of information. The result will be the creation of an in-app purchase with a unique ‘Product ID’. It is this ID that you will use while implementing your in-app purchase in LiveCode.

Syntax

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

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

Commands and Functions

To determine if in app purchasing is available use:

mobileCanMakePurchase()

Returns *true* if in-app purchases can be made, *false* if not. Throughout the purchase process, the AppStore sends **purchaseStateUpdate** messages to your app which report any changes in the status of active purchases. The receipt of these messages can be switched on and off using:

mobileEnablePurchaseUpdates

mobileDisablePurchaseUpdates

To create a new purchase use:

mobilePurchaseCreate productID

The *productID* is the identifier of the in-app purchase you created in iTunesConnect and wish to purchase. A purchaseID is placed in the result which is used to identify the purchase. To query the status of an active purchase use:

mobilePurchaseState(*purchaseID* **)**

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

- *initialized* – the purchase request has been created but not sent. In this state additional properties such as the item quantity can be set.
- *sendingRequest* – the purchase request is being sent to the store / marketplace.

- *paymentReceived* – the requested item has been paid for. The item should now be delivered to the user and confirmed via the **mobilePurchaseConfirmDelivery** command.
- *complete* – the purchase has now been paid for and delivered
- *restored* – the purchase has been restored after a call to mobileRestorePurchases. The purchase should now be delivered to the user and confirmed via the mobilePurchaseConfirmDelivery command.
- *cancelled* – the purchase was cancelled by the user before payment was received
- *error* – An error occurred during the payment request. More detailed information is available from the mobilePurchaseError function

To get a list of all known active purchases use:
mobilePurchases()
It returns a return-separated list of purchase identifiers, of restored or newly bought purchases which have yet to be confirmed as complete.
Before sending an your purchase request using the **mobilePurchaseSendRequest**, you can configure aspects of it by setting certain properties. This is done using:
mobilePurchaseSet *purchaseID, property, value*
The parameters are as follows:

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

Properties which can be set include:

- *quantity* – specifies the quantity of the in-app purchase to buy (default 1) (iOS)
- *developerPayload* – a string of less than 256 characters that will be returned with the purchase details once complete.

Can be used to later identify a purchase response to a specific request. Defaults to empty (Android)

As well as setting properties, you can also retrieve them using: **mobilePurchaseGet** *(purchaseID, property)*

The parameters are as follows:

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

Properties which can be queried include:

- *quantity* – amount of item purchased
- *productID* – identifier of the purchased product
- *purchaseDate* – date the purchase / restore request was sent
- *transactionIdentifier* – the unique identifier for a successful purchase / restore
- *developerPayload* – the developer payload value that was sent with the original purchase request (Android)
- *signedData* – a string containing detailed information about the purchase request response, in JSON format. This is signed by the Android Market using the private key application developer's publisher account, the public half of the key pair can then be used to verify that the message came from the Android Market. (Android)
- *signature* – the cryptographic signature of the signedData, in base64 encoding (Android)
- *receipt* – block of data that can be used to confirm the purchase from a remote server with the itunes store (iOS)
- *originalPurchaseDate* – for restored purchases – date of the original purchase (iOS)
- *originalTransactionIdentifier* – for restored purchases – the transaction identifier of the original purchase (iOS)
- *originalReceipt* – for restored purchases – the receipt for the original purchase (iOS)

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

mobilePurchaseSendRequest *purchaseID*
Here, *purchaseID* is the identifier of the purchase request. This command should only be called on a purchase request in the 'initialized' state.
Once you have sent your purchase request and it has been confirmed you can then unlock or download new content to fulfill the requirements of the in-app purchase. You must inform the AppStore once you have completely fulfilled the purchase using:
mobilePurchaseConfirmDelivery *purchaseID*
Here, *purchaseID* is the identifier of the purchase request. **mobilePurchaseConfirmDelivery** should only be called on a purchase request in the 'paymentReceived' or 'restored' state. If you don't send this confirmation before the app is closed, **purchaseStateUpdate**messages for the purchase will be sent to your app the next time updates are enabled by calling the mobileEnableUpdates command.
To instruct the AppStore to re-send notifications of previously completed purchases use:
mobileRestorePurchases
This would typically be called the first time an app is run after installation on a new device to restore any items bought through the app.
To get more detailed information about errors in the purchase request use:
mobilePurchaseError *(purchaseID)*

Messages

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

purchaseStateUpdate *purchaseID, state*
The state can be any one of the following:

- *initialized* – the purchase request has been created but not sent. In this state additional properties such as the item quantity can be set.
- *sendingRequest* – the purchase request is being sent to the store / marketplace
- *paymentReceived* – the requested item has been paid for. The item should now be delivered to the user and confirmed via the mobilePurchaseConfirmDelivery command
- *complete* – the purchase has now been paid for and delivered
- *restored* – the purchase has been restored after a call to mobileRestorePurchases. The purchase should now be delivered to the user and confirmed via the mobilePurchaseConfirmDelivery command
- *cancelled* – the purchase was cancelled by the user before payment was received
- *error* – An error occurred during the payment request. More detailed information is available from the mobilePurchaseError function