

## Engine Changes

**Important:** As of 5.5-dp-3, the default stack file format has been updated to 5.5. The previous default version (2.7) has been set as a legacy option in the “Save As” dialog. As such, the field updates added in 5.5 are now no longer considered experimental.

**Warning:** File Format changes to LiveCode 5.5 mean stacks created in LC 5.5 DP2 and above could have data loss if opened in older versions (field style related data)

### **The unicodeToolTip property (5.5 RC1)**

A new property unicodeToolTip has been added. This property allows unicode text to be displayed in tooltips and functions in exactly the same way as the toolTip, except that it expects a UTF-16 encoded string, rather than a native encoded string.

### **Paragraph hiding (5.5 DP3)**

There is a new *hidden* paragraph attribute. When set to true, the given paragraph will not be drawn nor take part in any vertical height calculations - essentially hiding it from view.

The paragraph’s content still remains in the text buffer and will be included in any requested chunks assuming it is in range.

When exported as a styledText array, if a paragraph is hidden then the corresponding paragraph array will have a *hidden* key with value true.

### **Updated htmlText (5.5 DP3)**

The htmlText content format has been updated. It is now a fully faithful representation of field content, and includes all the new paragraph styles introduced in 5.5. Note that there have been a few minor changes to the format relative to previous versions.

Sequences of paragraphs with the listStyle attribute set will be encased by appropriate ordered/unordered list tags - <ul> for unordered (bulleted) styles and <ol> for ordered (enumerated) styles. The paragraph (or paragraphs if the skip listStyle is present) associated with each list item is encased inside an <li> tag.

Each paragraph will be encased by <p>...</p> tags. The <p> tag can have zero or more of the following attributes: align, firstindent, leftindent, rightindent, spaceabove, spacebelow, tabstops, bgcolor, borderwidth, hgrid, vgrid, bordercolor, dontwrap, padding, hidden.

The character level styles applied to runs of text in paragraphs are represented using the following tags:

- textShift: if negative uses the <sup> tag otherwise uses the <sub> tag; in either case the value of the textShift property is present as a *shift* attribute.
- textFont, textSize, textColor, backgroundColor: combined into a <font> tag with one or more attributes *face*, *size*, *color*, *bgcolor*.
- textStyle: map to the <i>, <b>, <strike>, <u>, <condensed>, <expanded>, <threedbox>, <box> tags as appropriate.
- linkText: maps to an <a> tag with *name* attribute if no link textStyle set, or *href* otherwise.

- *metadata*: maps to a `<span>` tag with *metadata* attribute.
- *imageSource*: maps to an `<img>` tag with *src* attribute and *char* attribute. The *char* attribute contains the character the image replaces in the field.

The text encoding of the content is ASCII so any non-ASCII characters are encoded using the standard HTML entities as far as possible; any remaining non-ASCII characters being represented as numeric entities (`&#<d>;`).

### **Updated rtfText (5.5 DP3 – Updated 5.5 RC1)**

The rtfText content format has been updated to include the new paragraph attributes added in 5.5. In addition a number of issues have been resolved:

- font sizes now map correctly to points.
- links (linkText together with link textStyle) now appear as HYPERLINK fields.
- the *cb* and *chchpat* tags are now used to specify character-level background color (instead of highlight).

Note that rtfText is an effective format and is not fully faithful.

As of 5.5-rc-1, importing rtfText supports the new paragraph attributes.

### **Unlinking of text properties in objects (5.5 DP2)**

Similar to field chunks, in previous engine setting one of the *textFont*, *textSize*, *textStyle* or *textHeight* properties on an object would cause all four to be set if they had not already been previously; the values being inherited from the parent.

This behavior has also been changed in 5.5. The engine now allows all these properties to be set individually (*textHeight* has special rules - see below).

This change is persistent for stacks that are saved in 5.5 and then reloaded into 5.5 and later. If a stack saved in 5.5 is loaded into an earlier engine then the separation of these properties will be lost and it will appear as it would had the properties been set in that engine. (i.e. Any unset properties in the stackfile will transpire as properties set to the nearest ancestor's values).

Note that *textHeight* is always set when *textSize* is set, and if *textHeight* is not set then it will be calculated from the effective *textSize*. So, if you want to change the *textHeight* and *textSize*, set the *textSize* first, then set the *textHeight*.

### **Manipulating Unicode in objects (5.5 DP2)**

In previous engines buttons, groups and graphics could use unicode text by marking the *textFont* property with a “,unicode” tag; this would cause the engine to interpret the text and label properties as UTF-16.

This behavior has changed in 5.5. The engine now treats the unicode nature of the text and/or label properties of buttons, groups and graphics as an intrinsic property that cannot be altered by script. In particular, putting UTF-16 text into a button then settings its *textFont* property to “,unicode” (or “<font>,unicode”) will no longer have any effect.

To use unicode with these objects then use the following properties:

buttons: the *unicodeText* to set the menu description (if a menu); the *unicodeLabel* to the button label.

graphics: the *unicodeLabel* to set the graphic label.

groups: the *unicodeLabel* to set the group label.

To determine whether an object is currently storing its text and/or label as unicode there is a new read-only *encoding* property. This will return unicode if the engine is currently using UTF-16 to store the text/label, or native otherwise.

To fetch the current content/label of the object as native use the text or the label properties. This is a change from previous engines where these properties would return either native or UTF-16 text, depending on whether there was a “,unicode” tag in the (effective) *textFont*.

These changes are persistent for stacks that are saved in 5.5 and then reloaded into 5.5 and later. If a stack saved in 5.5 is loaded into an earlier engine then the *textFont* property of the object (and other text properties) will be set to have the *effective textFont* together with the (expected) “,unicode” tag

## New Field Features

### Flagged char ranges (5.5 DP1)

There is a character-level property flagged in fields.

Runs of characters that have this property set to **true** will be underlined with a dotted red line, flagging them as potentially needing attention. The purpose of this property is to allow easier implementation of on-demand spell-checking in fields.

The *flagged* property differs from other character properties in that it is considered extra information that is not part of the styling of the field. This means that its value does not appear in, and has no effect on, any of the (persistent) content formats (*styledText*, *rtfText*, *htmlText*) nor will it appear when content is copied or pasted.

To make it easier to handle the flagged property (since it does not appear in any persistent form), there is a field and field chunk level property *flaggedRanges*. This property returns a return-delimited list of character ranges on which *flagged* is set to **true**. The character ranges are relative to the first index of the chunk the range is requested for.

For example, consider the following text (underline indicates the flagged ranges):

Hello LiveCode World

Goodbye LiveCode World

With this we get:

the *flaggedRanges* of field => 7,14<cr>30,37

the *flaggedRanges* of line 1 of field => 7,14

the *flaggedRanges* of line 2 of field => 9,16

The *flaggedRanges* is both settable and gettable. For example, the following has no resulting effect:

set the *flaggedRanges* of line 2 of field 1 to the *flaggedRanges* of line 2 of field 1

Note: *The flagged property is useful as it gives a way of highlighting text which does not in any way effect the content. The flaggedRanges property is useful because it allows you to easily transfer the flagging between fields and portions of fields.*

## Chunk index conversion (5.5 DP1)

There are two field chunk properties *charIndex* and *lineIndex*.

These properties allow easy (and efficient) conversion between the character index at which a beginning of a line sits, and the line index in which a character sits.

The *charIndex* of a field chunk will return the character offset in the field of the start of the chunk.

The *lineIndex* of a field chunk will return the line offset in the field of the start of the chunk.

For example, consider the following text:

Hello World

Goodbye World

With this we get:

the *charIndex* of line 2 of field => 13

the *lineIndex* of char 7 of field => 1

the *charIndex* of word 2 of field => 7

Note that the *charIndex* property takes into account the Unicode nature of the chunk (if applicable) meaning that in some cases the *charIndex* of char 5 is not necessarily 5.

Note: *This is useful because there are many places where the engine gives you a field chunk in char or line form, but you might want the the opposite indices. (e.g. the *lineIndex* of the *selectedChunk* gives you the index of the line containing the selection).*

## Metadata char property (5.5 DP1)

There is a new character-level property *metadata* in fields.

This property can be set on runs of characters and is intended to allow applications to store data attached to character ranges in fields that has no effect on engine processing.

The value of the *metadata* property is limited to non-binary strings at present.

Note: *This is useful because it allows data to be associated with text in fields that moves automatically as the field is edited. While you could do something like this with the *linkText* property that already existed, setting the *linkText* has an effect on both the visual presentation and UI interaction of text in fields.*

## Individual textStyle manipulation (5.5 DP1)

The *textStyle* property of fields has been extended with array notation to allow easier manipulation of individual styles.

It is now possible to use expressions of the form:

the *textStyle*[<style>] of <chunk> of field

Where `<style>` is one of **bold**, **condensed**, **expanded**, **italic**, **oblique**, **box**, **threedbox**, **underline**, **strikeout**, **link**, **group**.

Such a property can be set to either **true** or **false** and will result in either that style being added (true), or removed (false) from the `textStyle`'s of each character in the range.

Getting the property will result in one of **true**, **false**, or **mixed** depending on whether the property is set on all chars in the chunk (true), unset on all chars in the chunk (false), or set on some and unset on others (mixed).

For example, consider the following text where the middle word is in italics:

Hello *LiveCode* World

Performing the following command will add bold style to the given range without affecting the setting of italic:

set the `textStyle`["bold"] of char 3 to -3 of field to true

Results in:

Hello ***LiveCode*** World

*Note: This is useful because previously there was no (straightforward) way to set text styles individually. Indeed, the `textStyle` property is a comma-separated list of styles applied to the character so if you set it to italic, it obliterates the bold setting (if present).*

## textChanged message (5.5 DP1)

The field will now dispatch a message *textChanged* whenever a user (or simulated user) action may have caused the content of the field to change.

In particular, the following actions will result in a *textChanged* message being sent:

- typing into the field (whether non-Unicode or Unicode text)
- using the **type** command to type into a field
- pasting text into a field (whether via built-in keyboard shortcut, or the **paste** command)
- cutting text from fields (whether via built-in keyboard shortcut, or the **cut** command)
- drag drop operations on fields

The message is sent immediately after the operation completes, but before a screen update has been requested - the corresponding update will occur at the end of the first command in the *textChanged* handler. This means that you can lock screen as the first line of the handler to delay the update (allowing you to modify the content of the field without any flicker).

To prevent potential for infinite recursion, calls to *textChanged* will not nest. That is to say that if a *textChanged* handler is being executed for a given field, another *textChanged* message will not be sent to it should one be triggered.

The *textChanged* message will be sent after messages such as *keyDown* and *pasteKey* but before messages such as *keyUp*.

*Note: This is useful because at the moment there is no simple way to react when something non-script based modifies the field, you have to hook into numerous messages. Indeed, some things (such as Unicode text entry) result in no notification at all.*

## Unicode-aware field chunks (5.5 DP1)

The field will now (internally) correctly compute character ranges for chunks regardless of whether the text in the chunk is native, Unicode or a mixture.

Essentially this means that you can set and get properties of chunks in fields without having to worry about handling unicode text specially.

For example, consider the following text:

Hello Кґыышфт World

Goodbye Γρεεκ World

Previously attempts to do things like:

set the `textStyle` of word 2 of line 1 of field to “bold”

set the `backColor` of line 2 of field to “red”

Would, generally, not work because the field would not take into account the (two-byte) unicode characters correctly.

It should be noted that care still needs to be taken when manipulating the text content of such chunks. The value of a chunk such as:

line 2 of field 1

Will still be the ‘mixed’ representation the engine has internally which cannot really be processed in script. If there is need to process the content of chunks then the `text` and `unicodeText` properties should be used:

the `text` of line 2 of field 1 => “Goodbye ????? World” (native text encoding)

the `unicodeText` of line 2 of field 1 => “Goodbye Γρεεκ World” (UTF-16 text encoding)

Notice that the `text` property converts the content to the native text encoding (MacRoman, Windows-1252, ISO-8859-1 depending on platform) with ‘?’ used for unconvertable characters; whereas the `unicodeText` property returns the content of the chunk as (uniform) UTF-16.

*Note: This is useful because it enables many operations on fields to be performed regardless of whether the field contains Unicode text. Previously it was (basically) impossible to do anything with field chunks if the field contained Unicode text.*

## styledText content format (5.5 DP1)

The field now has a new ‘content’ format - *styledText*. This is similar to `rtfText` and `htmlText` in that it provides a script-processable representation of the field’s content. It differs from these two formats in two ways: it is a fully faithful representation (set the `styledText` of field to the `styledText` of field results in no change to the field); it is array-based.

The `styledText` property returns a numerically-indexed array of paragraphs, each index representing each paragraph in the field in order:

`tStyledTextArray[1] = <first paragraph array>`

...

`tStyledTextArray[<n>] = <last paragraph array>`

Each paragraph array has up to two keys:

```
tParagraphArray["style"] = <array containing paragraph-level styles>
```

```
tParagraphArray["runs"] = <paragraph content array>
```

The style array contains the values for each of the paragraph styles set on that paragraph. The list of styles that are supported are: *textAlign*, *listStyle*, *listDepth*, *listIndent*, *firstIndent*, *leftIndent*, *rightIndent*, *spaceAbove*, *spaceBelow*, *tabStops*, *backgroundColor*, *borderWidth*, *borderColor*, *hGrid*, *vGrid*, *dontWrap* and *padding*.

The paragraph content array is a numerically-indexed array of runs, each index representing each run in the paragraph in order:

```
tParagraphContentArray[1] = <first paragraph run array>
```

```
...
```

```
tParagraphContentArray[<n>] = <last paragraph run array>
```

Each paragraph run array has up to three keys:

```
tRunArray["style"] = <array containing character-level styles for the run>
```

```
tRunArray["metadata"] = <metadata of the run (if present)>
```

```
tRunArray["text" (or "unicodeText")] = <text content of run>
```

The style array contains the values for each of the characters styles set on that run. The list of styles that are supported are: *textFont*, *textSize*, *textStyle*, *textColor*, *backgroundColor*, *linkText* and *imageSource*.

If a run has Unicode text in it then the run array will have a "unicodeText" key containing its content encoded as UTF-16. Otherwise, the run array will have a "text" key containing its content encoded in the native text encoding.

For example, take the following content consisting of two paragraphs:

Centered **Hello** World

Left-aligned **Hello** ЦЩКДВ

This would transpire as the following array:

```
1 =>
```

```
style = { textAlign => "center" }
```

```
runs =>
```

```
1 = { text => "Centered " }
```

```
2 =>
```

```
style = { textStyle => "bold" }
```

```
text = "Hello"
```

```
3 = { text => " World" }
```

```
2 =>
```

```
runs =>
```

```

1 = { text => "Left-aligned " }
2 =>
    style = { textColor => "255,0,0" }
    text = "Hello"
3 = { unicodeText => "ЦЩКДВ" }

```

[ For brevity, single element arrays are represented using { ... } notation ]

When setting the *styledText* property, the engine uses a very permissive algorithm to parse the arrays as follows:

```

parseStyledTextArray pStyledText
  repeat for each element tEntry of pStyledText
    if tEntry is a sequence then
      parseStyledTextArray tEntry
    else if tEntry has key "runs" then
      begin paragraph with style tEntry["style"]
      parseStyledTextRunArray tEntry["runs"]
      end paragraph
    else if tEntry is an array then
      append tEntry["text"] with style tEntry["style"]

parseStyledTextRunArray pRun
  repeat for each element tRun in pRuns
    if tRun is a sequence then
      parseStyledTextRunArray tRun
    else
      append tRun["text"] with style tEntry["style"]

```

The engine flattens any nested numeric arrays within the tree; then iterates through the result ignoring any empty entries. If an array has a 'runs' key it is treated as an independent paragraph; otherwise it is assumed to be a 'run' and the styled text it contains is appended to the current paragraph. Note that the 'text' field of a run can contain newlines the presence of which will cause a paragraph break at the appropriate points - if such a break is made, the paragraph attributes will be copied across the break.

*Note: This property is useful because it provides a easily manipulable representation of the field contents. In particular as it is array-based it allows much easier modification of specific parts of the field than having to manipulate the string-based *htmlText* and *rtfText*.*



## Hierarchical list support (5.5 DP1)

The engine now has support for hierarchical lists in fields. These are controlled with the *listStyle*, *listDepth* and *listIndent* paragraph-level properties.

The *listStyle* property determines what kind of index is displayed for the paragraph. It can be one of the following: **disc**, **circle**, **square**, **decimal**, **lower latin**, **upper latin**, **lower roman**, **upper roman** and **skip**.

The disc, circle and square settings cause the paragraph to be displayed with a bullet; while the decimal, lower/upper latin and lower/upper roman settings cause the paragraph to be displayed with its index written in the specified way (number, alphabetic or roman numerals). The index of a paragraph is computed as the number of preceeding paragraphs with the same listStyle before a paragraph is encountered with the same or lesser listDepth and different listStyle. In particular, this rule means that indexing continues after nested lists in the way you would expect.

The skip setting causes the paragraph to be displayed with an empty label - such paragraphs are ignored when computing the index of other paragraphs allowing it to be used to display multiple paragraphs in the same list item.

The *listDepth* property is a number between 1 and 16 which determines the nesting depth of the list. The depth of a list is used to compute the paragraph's list index as well as the indentation (if firstIndent is not set).

The *listIndent* property determines the amount to indent the content of the paragraph by for each level of depth. If the *firstIndent* property of the paragraph is set instead, the engine uses firstIndent together with leftIndent to determine the placement of the label and content - in this case, the listDepth is ignored.

For example:

- listStyle = disc, listDepth = 1, listIndent = 20 (so text starts at 20)
  - a) listStyle = lower latin, listDepth = 2, listIndent = 20 (so text starts at 40)
  - b) listStyle = lower latin, listDepth = 2, listIndent = 20 (so text starts at 40)
    - i. listStyle = lower roman, listDepth = 3, listIndent = 20 (so text starts at 60)
  - c) listStyle = lower latin, listDepth = 2, listIndent = 20 (so text starts at 40)

*Note: This is useful as it makes it easy to display (and allow the user to edit) nested lists with automatic labelling.*

## Paragraph level properties (5.5 DP1)

The engine now has support for paragraph-level properties. Some existing field-level properties have been lifted to the paragraph-level, alongside several new properties for controlling the display of paragraphs.

The available paragraph-level properties are as follows: *textAlign*, *listStyle*, *listDepth*, *listIndent*, *firstIndent*, *leftIndent*, *rightIndent*, *spaceAbove*, *spaceBelow*, *tabStops*, *backgroundColor*, *borderWidth*, *borderColor*, *hGrid*, *vGrid* and *dontWrap*.

The *textAlign* property allows the alignment of a paragraph to be set independently. It can be one of

**left, center, right.** If the property is not set on a paragraph, the setting is inherited from that of the field.

The *firstIndent* property determines how much to indent the first line of a paragraph. The indent is considered to be relative to the *leftIndent* setting of the paragraph so positive values shorten the first line whereas negative values lengthen the first line. [ Note that in the case of a paragraph with non-empty *listStyle*, the *firstIndent* property (if set) determines the offset of the label relative to the start of the text and (in that case) must be negative. ]

The *leftIndent* and *rightIndent* properties determine how much space to leave at either side of the paragraph before any border and content is rendered.

The *spaceAbove* and *spaceBelow* properties determine how much space to leave above and below the paragraph before any border and content is rendered.

The *tabStops* property allows tab-stops to be set on a per-paragraph basis. If the property is not set on a paragraph, the *tabStops* setting of the field is used.

The *backgroundColor* property allows the color of the content area (inside any paragraph border) to be filled (note that strictly speaking this property is not inherited, but the effect is the same as if it was as the background of the field is rendered before the paragraphs are so the background color at the field level will ‘show through’ to the paragraph if the paragraph has no background color).

The *borderWidth* property determines the width of the border to draw around the paragraph.

The *padding* property determines the amount of space between the border and the text content. In *vGrid* mode, each cell is reduced in width to accomodate the padding at either end of the cell (i.e. it provides padding in each cell in addition to around the line as a whole).

The *borderColor* property determines the color to use when drawing the border and grid (if either of the grid properties are set). The *borderColor* is inherited from the field if not set on the paragraph.

The *hGrid* property controls automatic generation of grid lines between paragraphs. If the *hGrid* is set to true on a paragraph (or inherits the true setting from the field) then an implicit 1 pixel wide border will be placed both above and below the paragraph. Additionally, the engine will elide borders of adjacent paragraphs which have *hGrid* set to true if their properties are compatible. Specifically, if two adjacent paragraphs have *hGrid* set, have the same *borderWidth* and no (or 0) *spaceAbove* and *spaceBelow*, a single pixel grid line will be placed between them rather than the normal border.

The *vGrid* property allows automatic generation of grid lines between tabs and cell clipping to be set on a per-paragraph basis. If the property is not set at the paragraph level it’s value is inherited from the field.

The *dontWrap* property allows line-wrapping to be set on a per-paragraph basis. If the property is not set at the paragraph level it’s value is inherited from the field.

*Note: These features are useful for obvious reasons. In particular, the paragraph-level indents, borders and back-colors make it much easier to develop well styled text content; and the paragraph-level hGrid, vGrid and tabStops enable simple tables to be embedded inbetween normal paragraphs.*

## **vGrid column visibility and tabWidths (5.5 DP1)**

It is now possible to hide individual columns when *vGrid* is in effect. The engine will skip any

columns which have an effective tabStop width of 0. For example, suppose you want a table-like layout with columns of 50 wide but with the first and third columns hidden. You can use the following tabStops:

0,50,0,50

This means the first column is of zero width (hidden), the second is 50 wide, the third is of zero width (hidden) then all subsequent columns are 50 wide.]

To help with control column visibility a new (synthetic) property tabWidths has been added. This property is similar to the tabStops property except that it returns the tab stops as a sequence of widths rather than as absolute positions. For example, the following are equivalent:

tabStops = 50,100,150,200,250,250

tabWidths = 50,50,50,50,50,0

*Note: This feature is useful as it allows ‘hidden’ information to be stored in simple tables in fields on a per-row basis. The tabWidths property makes controlling visibility of individual columns much easier, as the relevant item can just be replaced with 0.*

### **vGrid fixed width table mode (5.5 DP1)**

When vGrid is in effect, if the final tabStop has 0 width the engine treats the paragraph as a fixed width table, the width being the position of the final tabstop.

In this mode, the width of the paragraph is considered to be the (fixed) width and this width is used to do the following:

- compute the placement of the table in the line taking into account the textAlign property
- compute the background rectangle to fill with the backColor property
- compute the placement of the border (if any)

In this mode, rather than borders and background fill going from left margin to right margin, they will instead go from left of the table to right of the table.

For example, suppose you want a table-like layout with only 5 columns of 50 wide. You can use the following tabStops:

50,50,50,50,50,0

This will result in a table of width 250 pixels.

*Note: This feature is useful as it allows the number of columns to be explicitly specified (usually tabs continue for the entire width of the field) and then use the explicit width to place the table in the line allowing easy centering of simple tables within fields.*

### **Unicode handling changes (5.5 DP2)**

In previous engines, a sequence of unicode characters in a field was marked by having a “,unicode” tag in the textFont property of that run and the text of the field would contain mixed sequences of native encoded characters and UTF-16 encoded characters.

Both these behaviors have changed in 5.5. The engine now treats the unicode nature of a bytes in fields as an intrinsic property that cannot be altered by script. In particular, setting the text property of a field and then setting the textFont to “,unicode” will no longer have any effect.

Putting unicode text into a field can now only be done by setting the `unicodeText` property of the field (or a chunk of the field); using one of the content format properties such as `rtfText`, `styledText` or `htmlText`; or by a user action which causes unicode text to be entered into the field such as typing with a unicode IME.

The `textFont` property will no longer contain any ‘language’ tag, and setting the `textFont` of a chunk to one containing such a tag will result in the tag being stripped and ignored.

To determine whether a chunk in a field is encoded as either entirely unicode, entirely native or a mixture of the two there is a new read-only encoding property. [ Note that this property, currently, only reports whether text happens to be currently encoded in unicode, native or a mixture - not whether the text needs to be unicode, it is possible for runs of text that could be natively encoded to be encoded in unicode if that’s how it was placed in the field ].

As a result of these changes, the `text` property of a field returns the content in the native encoding. Similarly, when using a field as a destination or source in a chunk expression without using any of the ‘text’ properties, any chunk value will be natively encoded. i.e.

field 4 -- native

line 5 to 10 of field 3 -- native

the `unicodeText` of word 10 to 100 of field 2 -- unicode

Note that by separating out the unicode marker and `textFont` property it means that the `textFont` of a run can be set (and inherited!) independently of whether it is unicode or not. This change is persistent for stacks that are saved in 5.5 and then reloaded into 5.5 or later. If a stack saved in 5.5 is loaded into an earlier engine then, for runs which are unicode, the `textFont` property will be the inherited `textFont` together with the (expected) “,unicode” tag.

## Unlinking of text properties in fields (5.5 DP2)

In previous engines, setting one of the `textFont`, `textSize` or `textStyle` properties on a field chunk would cause all three to be set if they had not already been previously; the values being inherited from the parent. This meant that once you’d set one of the properties, the other two would no longer inherit from an ancestor.

This behavior has been changed in 5.5. The engine now allows these properties to be set individually, and thus they all inherit individually. i.e. If the `textFont` is set on a run of text in a field, changing say the field’s `textSize` property will result in the run of text with differing `textFont` inheriting the new size automatically.

This change is persistent for stacks that are saved in 5.5 and then reloaded into 5.5 and later. If a stack saved in 5.5 is loaded into an earlier engine then the separation of these properties will be lost and in that engine, any runs of text with one or two of the properties set will have all set (the others being inherited from an appropriate ancestor).

## Explicit line-breaks in fields (5.5 DP2 – experimental)

The engine will now interpret a `numToChar(11)` character in a field paragraph as an explicit line-break when the (effective) `dontWrap` of the paragraph is false. This allows multiple ‘lines’ to be displayed within a single paragraph.

The `formattedText` property has been updated to map any explicit line breaks to newlines.

Note that since the vGrid property turns `dontWrap` on for the paragraph, using the line-break char in table paragraphs will have no effect.

## Put ‘unicode’ form (5.5 DP2)

There is a new variant of the `put` command:

```
put unicode <expr> [ into | before | after ] <field chunk>
```

Here <expr> is any LiveCode expression that evaluates to a (binary) string; and <field chunk> is any chunk that resolves to a field (or portion of a field).

This command is equivalent to:

```
put <expr> [ into | before | after ] <field chunk>
```

Except that <expr> is treated as UTF-16 encoded text; rather than native encoded text.

For example, to build up a sequence of lines in a field where each line is Unicode you can do:

```
put unicode tMyUTF16Line1 after field 1
put return after field 1
put unicode tMyUTF16Line2 after field 1
...
```

## MacOS X Ask / Answer File Dialog

The MacOS X ask and answer file dialogs have been updated to use the Cocoa API.

## textStyle Property Array Variant (5.0.2 – experimental)

A new array variant of the `textStyle` property has been added allowing for access to individual textStyles independently from the others by specifying the required style as an array key.

```
the textStyle[<style>]
```

Here, style can be one of bold, condensed, expanded, italic, oblique, box, threedbox, underline, strikeout, link.

**Important: The features described above are experimental. This means that it may not be complete, or may fail in some circumstances that you would expect it to work. Please do not be afraid to try it out as we need feedback to develop it further.**

## Mac – revCapture video capture library (4.6.1 – experimental )

The revCapture library is begin developed as a replacement for revVideograbber – utilising the most up to date APIs on each platform, and eliminating many issues the previous library had (including problems introduced by Apple with iSight video-capture in 10.6.5 onwards).

At this time, the revCapture library is Mac-only, and is integrated as part of the revVideograbber external; it will be moved into its own external and become cross-platform over time.

While largely similar in function to revVideograbber, the revCapture library does have a number of differences:

- There are no configuration dialogs – instead, available inputs and codecs can be queried and configured through script.
- Previewing and recording can be started and stopped independently.
- Previewing takes place direct into a (configurable) image object, allowing the preview to be rendered with effects, blending, inks and interleave with other objects.
- Only supports output to QuickTime MOV files with a limited collection of preconfigured codecs.

The available handlers are described in the following sections.

***Note:** As it stands the revCapture library has only been tested with built-in iSight hardware, although there is no reason it shouldn't work with any hardware that QTKit supports.*

**Important: This feature is currently experimental. This means that it may not be complete, or may fail in some circumstances that you would expect it to work. Please do not be afraid to try it out as we need feedback to develop it further.**

## Session management

All capture operations take place within a capture session, which is managed using:

**revCaptureBeginSession**

**revCaptureEndSession**

Call **revCaptureBeginSession** to initialize the capture session, allowing the rest of the capture library to be used. When the capture session is done with, call **revCaptureEndSession**. This will release any hardware, and cancel any recording and previewing operations.

***Note:** Calling any of the revCapture functions without calling **revCaptureBeginSession** at some point before hand will cause a no session error to be thrown.*

## Input configuration

Unlike revVideograbber, configuration of the inputs to use for video/audio capture is done entirely through script rather than invoking standard dialogs.

### Querying available inputs

To query the currently available inputs capable of capturing audio or video data use:

**revCaptureListAudioInputs()**

**revCaptureListVideoInputs()**

These both return a newline-delimited list of the available inputs of the given type.

### Configuring the audio input

To configure the audio input to capture from in the current session, use:

**revCaptureGetAudioInput()**

**revCaptureSetAudioInput** *audioInput*

Where *audioInput* is one of:

- the name of an input returned by **revCaptureListAudioInputs()**
- *default*, indicating the default audio input device on the system should be used
- *none*, indicating that no audio should be captured

If setting an audio input fails because the new device could not be accessed, a *could not open input* error is thrown. If the current session could not connect to the new device, a *could not connect to input* error is thrown.

### **Configuring the video input**

To configure the video input to capture in the current session, use:

```
revCaptureGetVideoInput()  
revCaptureSetVideoInput videoInput
```

Where *videoInput* is one of:

- the name of an input returned by **revCaptureListVideoInputs()**
- *default*, indicating the default video input device on the system should be used
- *none*, indicating that no video should be captured

If setting an audio input fails because the new device could not be accessed, a *could not open input* error is thrown. If the current session could not connect to the new device, a *could not connect to input* error is thrown.

## **Compression configuration**

Similar to input configuration, choosing the codec to use when recording audio and video streams is done entirely through script.

### **Querying available codecs**

To query the currently available codecs for audio or video recording use:

```
revCaptureListAudioCodecs()  
revCaptureListVideoCodecs()
```

These both return a newline-delimited list of the available codecs of the given type.

### **Configuring the audio codec**

To configure the audio codec to use when recording, use:

```
revCaptureGetAudioCodec()  
revCaptureSetAudioCodec audioCodec
```

Where *audioCodec* is either *none* to indicate that no audio compression should be performed, or the name of a codec returned by **revCaptureListAudioCodecs()**.

If the given codec name is unrecognized, or is not suitable for audio compression then an *invalid*

*codec for media type* error is thrown. If recording is currently happening, a *recording in progress* error is thrown.

### **Configuring the video codec**

To configure the video codec to use when recording, use:

```
revCaptureGetVideoCodec()  
revCaptureSetVideoCodec videoCodec
```

Where *videoCodec* is either *none* to indicate that no video compression should be performed, or the name of a codec returned by **revCaptureListVideoCodecs()**.

If the given codec name is unrecognized, or is not suitable for video compression then an *invalid codec for media type* error is thrown. If recording is currently happening, a *recording in progress* error is thrown.

## **Preview management**

### **Configuring the video preview image**

In order to be able to see a video preview of a capture session, the long id of an image object to target needs to be provided. To manage this use:

```
revCaptureGetPreviewImage()  
revCaptureSetPreviewImage imageLongId
```

Where *imageLongId* is the long id of an existing empty image object to use; or empty to turn off the preview image.

Setting the preview image will lock the corresponding object meaning it can't be resized or have it's content modified by anything apart from the capture session. If the image cannot be found, locked, or an offscreen buffer initialized a suitable error will be thrown.

### **Configuring the audio preview volume**

To configure the volume of the audio preview of a capture session use:

```
revCaptureGetPreviewVolume()  
revCaptureSetPreviewVolume newVolume
```

Where *newVolume* is an integer between 0 (no audio preview) and 100 (audio preview at maximum volume).

### **Previewing a session**

To start or stop a preview from running use:

```
revCaptureStartPreviewing  
revCaptureStopPreviewing
```

The status of the preview is independent to that of recording, meaning that you can preview without recording, and record without previewing.



If a preview of the session could not be initialized when calling **revCaptureStartPreviewing** a *could not connect to output* error is thrown.

**Note:** Starting to record may have an effect on the video preview's aspect and frame size as the video capture pipeline is optimized by the system based on the needs of the recorded output.

## Record management

### Configuring the output file

To configure the filename to use when recording a capture session use:

```
revCaptureGetRecordOutput()  
revCaptureSetRecordOutput recordFilename
```

Where *recordFilename* is an (absolute) path to the file to use for recording.

If recording is currently in progress, a *recording in progress* error is thrown.

### Configuring the maximum frame size

To configure the maximum size of a frame in the recorded file use:

```
revCaptureGetRecordFrameSize()  
revCaptureSetRecordFrameSize maxWidth, maxHeight
```

Where *maxWidth* and *maxHeight* specify the maximum size of a frame (in pixels) that should be generated in the (compressed) output. To use the default (optimal) frame size for the current settings, specify both width and height as 0.

**Note:** These act as a hint to the pipeline, and the actual size of the frames in the output file may be smaller than this, or have a different aspect ratio.

### Configuring the maximum frame rate

To configure the maximum frame rate to aim for in the recorded file use:

```
revCaptureGetRecordFrameRate()  
revCaptureSetRecordFrameRate maxFrameRate
```

Where *maxFrameRate* is the number of frames per second to aim for. To use the default (optimal) frame rate for the current settings, specify the rate as 0.

**Note:** This acts as a hint to the pipeline as to the rate to aim for. The resulting frame rate in the recorded file may be less than, or more than this setting.

### Starting recording

To start recording the current session use:

```
revCaptureStartRecording
```

This configures the session with previously selected record options and codecs, deletes the currently specified output file (if present) and starts the recording process. It has no effect if the session is already being recorded.

If recording starts successfully, **the result** is empty.

If an error occurs when the output file is being prepared for recording, **the result** will contain *recording failed*.

If the session could not be configured for recording, a *could not connect to output* error is thrown.

### **Stopping recording**

To stop recording the current session use:

#### **revCaptureStopRecording**

This finishes outputting any pending samples to the output file and stops recording. It has no effect if the session is not currently being recorded.

If recording finished successfully, **the result** is empty.

If an error occurs while trying to finish writing the output file, **the result** will contain *recording failed*.

### **XML text node access (4.6 – experimental)**

Support has been added to some revXML functions for manipulating text nodes. Consider the following XML fragment:

```
<summary>
```

```
    Removes a <keyword>message</keyword> that was queued with the
<command>send</command> command and is waiting to be sent.
```

```
</summary>
```

This has the following structure as an XML tree:

```
ELEMENT(summary)
  TEXT(Removes a)
  ELEMENT(keyword)
    TEXT(message)
  TEXT( that was queued with the )
  ELEMENT(command)
    TEXT(send)
  TEXT( command and is waiting to be sent.)
```

Notice that the named XML nodes, are interspersed with (essentially unnamed) nodes containing text content. These (previously unaccessible) nodes can now be accessed (with some functions) by using a path of the form:

```
<parent>/[<index>]
```

Here this references the <n>th text node under <parent>. For example, the above tree has the following accessible nodes:

```
summary/[1]
```

```
summary/keyword
summary/keyword/[1]
summary/[2]
summary/command
summary/command/[1]
summary[3]
```

To access text content of a node simply use the *revXMLNodeContents* function with this extended path format. Note that the previous behavior is preserved – if you specify a node with a '[n]' suffix, the text content of the node will be returned (if it is a text node). (i.e summary/keyword and summary/keyword/[1] are the same thing to *revXMLNodeContents*).

The following functions have been augmented with an additional (optional) parameter *incText*:

```
revXMLFirstChild(docId, [ incText ])
revXMLNextSibling(docId, nodePath, [ incText ])
revXMLPrevSibling(docId, nodePath, [ incText ])
revXMLChildNames(docId, nodePath, delimiter, filter, incCounts, [ incText ])
```

Here if *incText* is specified and is *true*, the functions will include text nodes in their processing.

For example, this allows you to loop over all nodes *including text nodes* using something like:

```
local tCurrentNode
put revXMLFirstChild(tDocId, tParentNode, true) into tCurrentNode
repeat while tCurrentNode is not empty
  ... use tCurrentNode ...
  put revXMLNextSibling(tDocId, tCurrentNode, true) into tCurrentNode
end repeat
```

**Important: This feature is currently experimental. This means that it may not be complete, or may fail in some circumstances that you would expect it to work. Please do not be afraid to try it out as we need feedback to develop it further.**

## ***Ordered and unordered lists (4.6 – experimental)***

### **Properties**

Experimental support has been added to the field for simple, single level, ordered and unordered lists. To make a paragraph display as an element in a list use the *listStyle* property:

**set the listStyle of line *lineIndex* of field to style**

Where *style* is one of:

- *disc* – the paragraph is rendered as an element in an unordered list with the standard bullet character as marker (U+2022).

- *circle* – the paragraph is rendered as an element in an unordered list using the (unicode) character U+25E6 as marker.
- *square* – the paragraph is rendered as an element in an unordered list using the (unicode) character U+25AA as marker.
- *decimal* – the paragraph is rendered as an element in an ordered list, the label using standard (Arabic) decimal numerals. i.e. 1, 2, 3, 4, etc.
- *lower latin* – the paragraph is rendered as an element in an ordered list, the label using lowercase (Latin) letters. i.e. a, b, c, ..., aa, ab, etc.
- *upper latin* – the paragraph is rendered as an element in an ordered list, the label using uppercase (Latin) letters. i.e. A, B, C, ..., AA, AB, etc.
- *lower roman* – the paragraph is rendered as an element in an ordered list, the label using lowercase Roman numerals. i.e. i, ii, iii, iv, ..., etc.
- *upper roman* – the paragraph is rendered as an element in an ordered list, the label using lowercase Roman numerals. i.e. I, II, III, IV, ..., etc.

For both ordered and unordered list the marker is placed at the first tab-stop, and the content of the paragraph is placed (and wraps) at the second tab-stop.

For ordered lists the index of the item is determined by the number of preceding paragraphs with the **same** *listStyle* property.

Setting the *listStyle* property of a paragraph to empty causes it to revert to a normal (non-list item) paragraph.

## htmlText format

Paragraphs with a non-empty *listStyle* present themselves in *htmlText* wrapped with `<LI>` tags. Sequences of such paragraphs with the same *listStyle* are bracketed by `<UL>` or `<OL>` tags. These tags take a *type* attribute, matching the (legacy) HTML attribute of the same name:

- *disc* → *disc*
- *circle* → *circle*
- *square* → *square*
- *1* → *decimal*
- *a* → *lower latin*
- *A* → *upper latin*
- *i* → *lower roman*
- *I* → *upper roman*

For example:

1. Numbered Item 1
  2. Numbered Item 2
- Bulleted Item 1
  - Bulleted Item 2

```
<ol type='1' >
<li><p>Numbered Item 1</p></li>
<li><p>Numbered Item 2</p></li>
</ol>
<ul type='disc'>
<li><p>Bulleted Item 1</p></li>
```

```
<li><p>Bulleted Item 2</p></li>
</ul>
```

## rtfText format

Paragraphs with a non-empty *listStyle* are appropriately marked in *rtfText* using both the (legacy) *pn* family of paragraph numbering tags and also with the new *listtable* tags.

By using both sets of tags a reasonable degree of interoperability is achieved with both TextEdit (and other Cocoa applications) on Mac, and Word and WordPad on Windows.

**Note:** *Unfortunately, OpenOffice does not have particularly good rtf import / export capabilities (it doesn't even round-trip correctly through itself!) and thus copying / pasting of lists between LiveCode and OpenOffice will not work reliably or correctly.*

## plainText format

Paragraphs with a non-empty *listStyle* are appropriately exported in plain text form when using the *plainText*, *unicodePlainText*, *formattedText* and *unicodeFormattedText* properties.

For example, the above example would be rendered (for *plainText*) as:

```
<tab>1.<tab>Numbered Item 1<return>
```

```
<tab>2.<tab>Numbered Item 2<return>
```

```
<tab>•<tab>Bulleted Item 1<return>
```

```
<tab>•<tab>Bulleted Item 2<return>
```

## Persistence

As it stands the *listStyle* property does not save into the stack-file and will not do so until version 5.0 when the file format is revised (for various technical reasons, it is not possible to add this as a saveable property with the current stackfile format).

It is recommended that *htmlText* be used to save the content of fields in custom properties, for restoration on reload.

**Important: This feature is currently highly experimental. This means that it may not be complete, fail in some circumstances that you would expect it to work, or change considerably before becoming final. Additionally, it might have problems or gotchas that make it significantly harder to use than other LiveCode features at this time.**

## Runtime execution stack configuration (4.5.1 – experimental)

In order to be able to more reliably control the maximum level of recursion, a new global property **stackLimit** has been introduced.

This property allows a script to set (in bytes) the maximum size of the (runtime) stack the engine uses for recursive computation. A change in the setting will only take effect when all currently executing handlers complete, and at this time the stack size limit will be reconfigured to the given limit, or the nearest amount to it depending on available memory.

The *stackLimit* currently in effect can be fetched using **the effective stackLimit**.

The *recursionLimit* property is now bounded by the *stackLimit* – attempts to set the *recursionLimit*