Line Endings in LiveCode

by Richard Gaskin

When newcomers get started with LiveCode they soon discover that LiveCode's handling of line endings in text is unique, at times perhaps even confusing. Hopefully this article will help provide an understanding of why they're handled as they are and how to work with them effectively.

Background

In short, we can blame HyperCard. :)

LiveCode is a unique contribution to the world of programming languages, but it's also part of a long line of related languages that include HyperTalk, SuperTalk, OpenScript, MediaTalk, and others, collectively known as "xTalks".

HyperTalk was the name of the scripting lanugage of Apple's HyperCard, and while it arguably borrowed elements from SmallTalk, Pascal, and Silicon Beach Software's *World Builder*, it's generally regarded as the "mother tongue" of the xTalk family of lanugages.

So as with many conventions adopted by more modern xTalks, to understand the roots of how LiveCode handles line endings we need to go back and take a look at HyperCard.

Presuming that most scripters would be reading plain text files for display in a field, when reading a file in HyperCard it always translated NULL bytes to spaces and line endings to old-syle Mac line endings, ASCII 13. Accordingly, the xTalk constants *CR* and *return* became the most common way we refer to the line ending character.

When SuperCard was born in '89, their team decided that when you're reading a file you probably want the file's actual contents, rather than some permutation of them as HyperCard delivers. In HyperCard you needed an *external* (a compiled code module written in a lower-level language) to do binary reads, but SuperCard did them out of the box.

LiveCode was born on Unix (known then as "MetaCard"), where ASCII 10 is the line ending, and accordingly it uses the Unix line ending convention internally.

NeXT OS was a certified Unix, and now that the latest version of NeXT is OS X, the Mac platform now also uses Unix line endings.

For legacy reasons many Mac apps handle both ASCII 10 and ASCII 13 interchangeably in terms of display, and below we'll see that LiveCode's behavior works similarly to provide those conveniences across all of the platforms it supports

Working with Line Endings

LiveCode provides the automatic text transformation HyperCard provided by default, but goes beyond both HyperCard and SuperCard in letting you choose. HyperCard only read in what we call "text" mode, SuperCard only reads in what we call "binary" mode, but LiveCode provides both.
For example, to read a file in a way that allows the LiveCode engine to automatically convert line endings we'd use:

```
open file tMyFilePath for text read
read from file tMyFilePath untl EOF
put it into tVar
close file tMyFilePath
```

The "text" modifier works for *write* and *append* as well.
And since this is the default behavior, we don't even need specify the mode, and can simply use this instead of the first line of the example above:

```
open file tMyFilePath for read
```

We can also use the convenient URL syntax for this:

```
put url ("file:" & tMyFilePath) into tVar
```

If you want to preserve the file's contents exactly as they are you can specify reading or writing in *binary* mode:

```
open file tMyFilePath for binary read
```

Or using the URL syntax:

```
put url ("binfile:" & tMyFilePath) into tVar
```

The transformation that occurs with the default "text" mode turns out to be far more useful in LiveCode than it was in HyperCard, since HyperCard was for Mac only but LiveCode needs to run on many different platforms, each of which uses their own line endings. By providing "text" mode by default, your app can read and write text files and display their contents appropriately regardless which platform it's running on.
Given that LiveCode was born on Unix, it's not surprising that it uses the Unix convention for line endings internally, whether in fields or when using chunk expressions on variable contents.
You can of course use your own *lineDelimiter* if you choose to keep your data in a platform-specific format, or let LiveCode handle it automatically for you with reads and writes in "text" mode to make the

line endings used internally consistent across platforms, or even use the *replace* command where needed to handle any of this yourself.
When we consider the platform-independent nature of LiveCode, all of this isn't so much a problem as it is just one of the many things we need to learn when we branch out of the one OS we use most of the time to deploy to others we're less familiar with.
LiveCode provides the commands to let us handle line endings however we choose, more than any other xTalk, but it can't anticipate all the ways we may need to work with files so how we use them is up to us.

Gotcha alert: CR <> numToChar(13)

Perhap the biggest confusion when working with text in LiveCode stems from the mapping of the constants *CR* and *return* to ASCII 10, which is more comonly known as lineFeed, while the true ASCII definition of CR is 13. In LiveCode *CR*, *return*, *LF*, and *lineFeed* are synonymous.
This is done to preserve compatibility with the many other xTalk languages that provide the *CR* and *return* constants. If these constants weren't mapped as they are, more than twenty years of code written in other xTalk dialects would fail in LiveCode.
Given LiveCode's multiplatform nature, it may be helpful to think of the *CR* and *return* constants in the generic sense of being line endings.
When working with text it rarely matter what the actual ASCII value is, but if you're working with binary data it will often matter very much, and you can distinguish true ASCII 13 charaters using the *numToChar* function:

```
put numToChar(13) into tMacLineEnding
open file tMyFilePath for binary read
read from file tMyFilePath until CR
if the last char of it is tMacLineEnding then
   answer "This data was likely created on a Mac"
end if
close file tMyFilePath
```

While this mapping of the *CR* and *return* contants to ASCII 10 can be confusing at first, it's just one of the gotchas that every language has, borne of its unique history and place in the world. Although it takes some getting used to at first, every language has its oddites; at least we're not losing a million programmer-hours every year to tracking down bugs related to the difference between "=" and "==". ;)