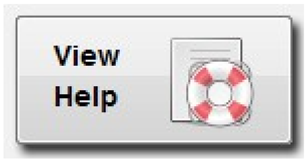
# WordLib

*The Word Processing Import Library for LiveCode*

**Version 2.3 User Guide**

## Viewing this document

If you're not viewing this user guide within the LiveCode IDE, you can! Just open the **Try WordLib** stack (if you're not there already) and press the **View Help** button.



You can also view this document with **OpenOffice**, a free word processor.

## About WordLib

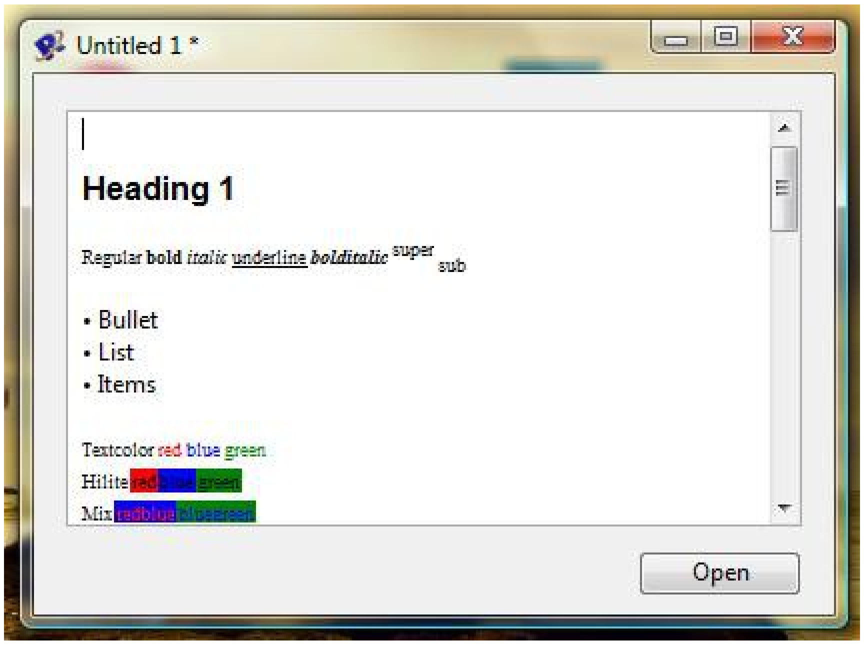
**WordLib** empowers your LiveCode projects to import word processing documents in a variety of formats, currently including **Microsoft Word** and **OpenOffice**.

With a single line of scripting code (two words) you can import a document, and that includes showing the user a file selection dialog and checking for errors. Thus, you and your end-users can easily open files created in word processing for display or editing. It's incredibly easy for beginners to get started, while advanced users have plenty of options to control the process.

You can also bundle documents along with your applications, so that users can read your help files or other documents either within your software or in a word processor as they prefer. (Like this **User Guide** you're reading.) Or you can store documents right inside your stack, even if they include images! Your word processor can be used like an external editor for LiveCode fields.

**History**:

Version **2.0** adds many new text formatting features, Android support, and easier Mobile testing.
Version **1.5** added iOS support and easier library installation.
(See Change Log file for detailed version history.)



Heading 1

Regular bold italic underline bolditalic super sub

- Bullet
- List
- Items

Textcolor red blue green

Hilite red blue green

Mix redblue bluegreen

Imported documents are displayed in your LiveCode fields and can be used normally, just as you use any other fields and field contents. **WordLib** provides support for many formatting features, from text fonts, colors, and sizes to bold/italic/underline to superscript and subscript, bullet lists, images, hyper links, footnotes, and even tables!

**WordLib 2.3** is recommended for LiveCode 8/9/10 or later. (Also LC 6.7.)
**WordLib 2.0** was recommended for LiveCode 5.5 or later.
**WordLib 1.6** is recommended for LiveCode/Revolution 5.0 or earlier.

**WordLib** is the real thing. It does all its import magic with pure native **RunRev** scripting code written by yours truly, giving you maximum power and compatibility!

(That includes native legacy .doc support!)

## Registration and trial

You can try out **WordLib** in the LiveCode programming environment. The demo won't time out, and most features are available. The demo is intended for personal, noncommercial use.

For standalone applications and any other non-personal or commercial uses (such as freeware, shareware, commercial software, consulting, use at work, workflow, for organization, for school, etc.) you'll need to register.

If you find **WordLib** helpful and really use it (commercially or otherwise) please register to support the project! This will help to add more features, more document formats, document export, and so on; and the support of registered users makes all this possible and is greatly appreciated.

To register or get the latest version of **WordLib**, go to:

http://livecodeaddons.com/wordlib.html

or click the **Purchase** button in the **Try WordLib** stack.

After you purchase, you'll get access to extra features (including saving documents right inside your stack with images), and the ability to use **WordLib** in your own standalone applications.

(See the "**After Registration**" section further down in this document!)

Upgrades: Major versions (2.3, 2.4, 3.0) cost $49 USD each.
Any minor updates to a version are free.
The current development goal is one major update release per year.

*(Policies and prices quoted here may change.)*

## What can I do with WordLib?

With **WordLib**, you can easily:

- Let users import documents for display or editing
- Use your word processor as an external editor for your field contents
- Create help files (like this one) in a word processor to display in your software
- Bundle documents as separate files or store them right inside your stacks
- Add import capability for building your own word processor
- Take advantage of new and popular file formats
- Spruce up your LiveCode field contents with better styles and formatting
- Easily display images in fields, or even tables
- Create documentation that's easier to read and more useful
- Use defined styles in Word to make formatting changes more efficient

## Document file formats

**WordLib** is a library for importing word processing documents in a variety of formats, currently including **Microsoft Word** and **OpenOffice**.

Most word processors (software applications for composing and formatting styled text documents) have their own file format and/or the ability to save in other popular formats. A **file format** is the way a document's text and related information (styles, formatting, images, etc.) are saved to your computer.

Back in 2007, Microsoft introduced **.docx** as the standard and default file format for **MS Word**. This will form the basis for Word's documents from this point on. (**WordLib** started as a project to import this new format.)

**WordLib** also handles the **.odt** format used by **OpenOffice**, a widely-used free word processor. (This User Guide is written in **OpenOffice**.)

**WordLib** supports many of the text styles and formatting found in your word processing documents. Here's a list of currently supported formats:

| | | |
|---|---|---|
| ● .docx, .docm, .xml | (Word 2007+) | extensive support |
| ● .odt | (OpenOffice) | strong support |
| ● .xml | (Word 2003) | strong support |
| ● .doc | (Word 97, etc.) | limited support |

I hope to add more formats and stronger support for existing formats in the near future.

## Conventions

Before we begin, a couple of notes on this documentation:

Tradition and taste in writing have dictated that punctuation always goes inside quotation marks, but when dealing with programming code and referencing precise names and values rather than conversation, it's very important to differentiate to the smallest detail between what's included and what is not, because any extra character could result in an error. In this guide, periods and commas will be placed outside quotation marks, so that what is quoted can be relied upon as precise information.

The **Control** key on Windows and the **Command** key on Macintosh serve many of the same purposes, and many shortcut key combinations use these. The same situation exists with **Alt** on Windows and **Option** on Macintosh. Meanwhile, traditionally Macs had one mouse button and used **Control-click** in situations where Windows users and newer Mac users **right-click**. Rather than write both variations each time an action is described, I will use (for example) **Control-C** for copying and **right-click** for contextual menus, and you'll know that **Command-C** and **Control-click** are the counterparts.

(Also, different releases may focus on one platform or feature more than others, and the same is true for this User Guide, so let me know if you find parts that need updating!)

## Getting started

For a quick start to try out the library for the first time and see how it works with importing documents, your best bet is the **Try WordLib** test stack. Just load it, choose a file to import, and have a look at the results. If you wish, you can also look at the scripts contained in the "**Import Document"** and "**View Help**" buttons *(Control-Alt-click)* and the card script *(Control-Shift-C)*. However, if you want a thorough and easy-to-understand explanation, this **User Guide** you're reading now is the right place to be.

**WordLib** runs on Windows, Mac, Android, and iOS!

This guide will help you when you are ready to learn how to use **WordLib** to import documents in your own projects. It does so at a comfortable pace, assuming that you do know a few basics about using the LiveCode programming environment, but explaining things thoroughly as we go.

For **experienced** LiveCode programmers, the following sample script may tell you all you need to know to get started, and you can scan down further in this guide to learn about more of the options. Don't forget to come back later and "RTM" if you need more info!

For **everyone else**, we'll take a look at the sample script as an exhibit, but then we'll slow down and discuss everything you need to know, right from the very beginning.

(Glancing at the brief length of the first two parts of the example, you can see that it isn't very scary, because it only takes two lines of scripting code, placed in the appropriate handlers, to load the library and import a document, and that's all you need to start using **WordLib**. And if you're wondering what in the world a handler is, don't worry; we'll discuss that further down.)

## Sample script

In the stack or first card script:

```
on preOpenCard
  -- load the library file (assuming it's in "the folder")
  start using "wordlib.rev"
end preOpenCard
```

In an import button (Option A, the easy method):

```
on mouseUp
  -- import document (auto file dialog & error check)
  importDocumentToField "docField"
end mouseUp
```

The above works on Mobile too! You can optionally specify a folder, which on Desktop provides the initial directory for the file chooser, and on Mobile specifies the folder containing documents. This can be an absolute path, or use "#/path" for a relative path from the current stack (Desktop) or the app folder (Mobile).

```
on mouseUp
  -- import document (with starting folder)
  importDocumentToField "docField","#/Documents"
end mouseUp
```

In an import button (Option B, for power users):

```
on mouseUp
  -- show file dialog
  answer file "Choose a Microsoft Word or OpenOffice document:"
  if the result is not "cancel" then
    put it into theFile
```

```
      -- import file to variable
      put htmlTextOfDocument(theFile) into theImport
      -- error check
      if item 1 of theImport is "error" then
        -- display error message
        answer item 3 to -1 of theImport
      else
        -- display document in field
        set htmlText of field "docField" to theImport
      end if
    end if
end mouseUp
```

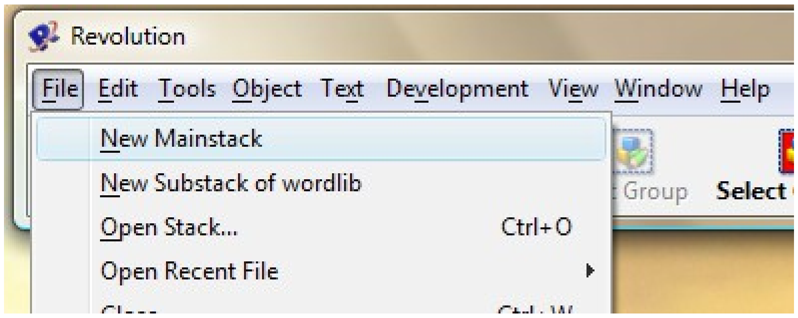
In an import button (Option C, another easy method):

```
on mouseUp
  -- import document (located in your app's folder)
  importBundledDocument "Welcome.docx","docField"
end mouseUp
```

(On Mobile, by copying the desired document files to your app bundle via the Standalone Settings, just pass the file name as in this example.)

## Creating a stack

If you haven't already created a stack for your project, please do so now. *(File menu, "New Mainstack".)* See LiveCode's documentation if you need help.



*(Yes, I'm keeping a few Revolution-era screenshots in this Guide for historic flavor! WordLib combines support for the very latest LiveCode field features with classic and easy-to-use Rev/LC techniques.)*

## Loading the library

Before you can do anything with a library such as **WordLib**, you need to load it. If you are using the **Try WordLib** stack, the library is already loaded, so at the moment you can use it without further effort.

However, you'll want your own project stacks to load the library themselves and work independently, so take a moment sooner or later to learn the art of LC library loading, explained below! It's easy.

What's a library, by the way? That's standard LiveCode terminology for a handy collection of scripting code (commands and functions) that you can use to add features and accomplish tasks in your own projects without doing the heavy lifting of programming all those features by yourself.

Loading **WordLib** into memory so that LiveCode can use it is simple. You can either embed the library as a substack and reference it by name, or keep it in a separate file and reference its filepath.

Each approach has its strengths! Let's look at the first:

## *THE LIBRARY FILE APPROACH*

My favorite approach is to keep the **WordLib** library in a separate file. This makes maintenance and upgrades a breeze.

**Try WordLib** uses this method itself, so its card script serves as a handy example!

Here's the syntax for loading **WordLib** as a file:

```
start using "wordlib.rev" -- file
```

However, LiveCode needs to know where the library is located on your computer in order to open it. The above line of code will work if **wordlib.rev** is already in the **defaultFolder** (also known as **the folder)** which specifies the current folder where LiveCode looks for files.

If you have **the folder** set to the location containing the **WordLib** library, you're all ready!

But what if you need to locate the library file? No problem. LiveCode's built-in path function includes the folder containing your project stack. Copy wordlib.rev into that folder, and the script looks like this:

```
put specialfolderpath("resources") & "/wordlib.rev" into f
start using f
```

(Always place cclib.rev in the same folder as  wordlib.rev.)

Now all you need is to load the library once in each project with that line of code you saw above:

```
start using "wordlib.rev"
```

To make this line of code do its job, you'll need to place it in an appropriate handler. (A "handler" handles a message which LiveCode sends to your project, and this often corresponds to an event that was caused by the user or the computer's operating system.)

A good candidate is the **preOpenStack** handler, which LiveCode calls when your stack has just been opened. Put this in the script of the first card in your stack. *(Control-1,Control-Shift-C.)* Type "on preOpenStack" and press **Enter**, then type the command we're using to load the library. We end up with something like this:

```
on preOpenStack
  start using "wordlib.rev"
end preOpenStack
```

Press the **Apply** button and close the script editor window (or press the numeric keypad **Enter** key twice) and you're finished with loading the library in your project. Now you can skip down to the next heading to prepare a field for importing documents, or keep reading in this section to learn a little more about the intricacies of library loading.

By the way, there's a synonym for the "start using" command:

```
library stack "wordlib.rev"
```

It does the same thing; you can use either command according to taste.

Later on, when you build a standalone application, make sure that a copy of **wordlib.rev** is included in the appropriate location. You can do that yourself, or let LiveCode's standalone builder handle the files for you.

The **defaultFolder's** initial location will become that of the standalone executable when it starts up, and specialfolderpath("resources") also will be relative to the engine.

Be sure to test the standalone (and register WordLib!) before distributing your software.

Using this method, as long as you put a copy of the library in the same folder as your project stack file, or tweak the code to use a subfolder such as Extensions, you're all set.

## ***THE SUBSTACK APPROACH***

Here's the other method!

A substacked library is very easy for LC to find/load at runtime, but requires a little more maintenance for installation and upgrades: you'll need to copy wordlib.rev and cclib.rev as substacks in your project.

*(Now for the tricky part: In version 2.3 I've removed the **Try WordLib** stack's handy **Install WordLib** button; it's due for an overhaul. Look for that tool to reappear in 2.4 or another future version!)*

Here's the script for loading **WordLib** once it's embedded in your stack:

```
on preOpenStack
  start using "wordlib" -- substack
end preOpenStack
```

The trick to remember, though, is to place this script in the **card** script of the first **card** of your project stack.

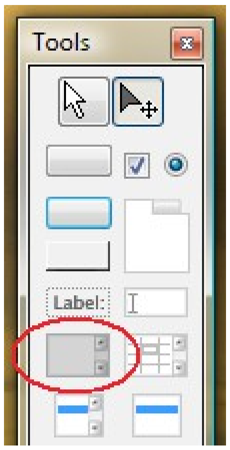
You may be thinking, since this is preOpen**Stack**, why use the **card** script? That's because if you put it in the **stack** script, it can also be triggered again each time any other substack opens. In the card script, it executes only once, just as we want it!

*(WordLib does not automatically insert code into your scripts; I believe that approach can be very problematic. You should maintain complete control over your code.)*

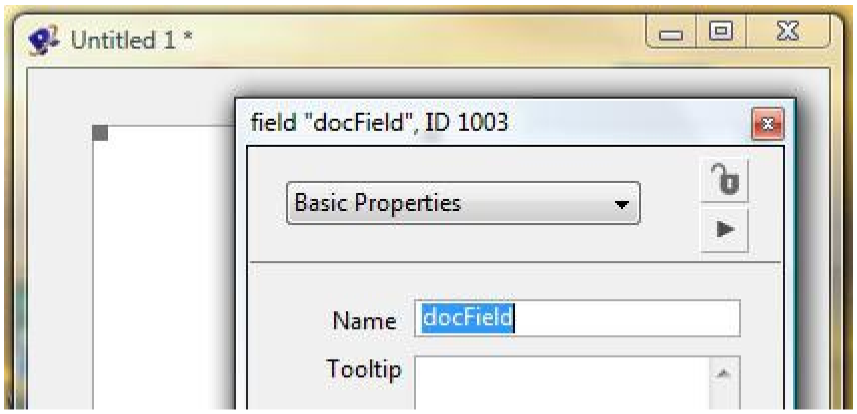
## Creating a field

Once you have the library in memory, you can use it to import documents into fields. But obviously, first you need a **field**.

To create a field to hold the documents, drag a field from the LiveCode **tool palette**, or double-click the tool palette icon to place a field. Then you can position and resize the field as desired.



Unless you're pretty sure that you'll have only one field on the card, it's a good idea to give it a **name**. Double-click the field and type a name into the **property inspector**.



field "docField", ID 1003

Name: docField

(For the purposes of this help file, let's assume that "**docField**" is your field's name. However, you could just as well call it "**doc**" or "**f1**" or "**displayField**" and so on.)

Other options you may want to consider:

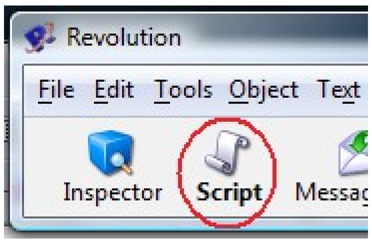
- Turning off **fixed line height** will result in better display since many documents contain varied text sizes and images.
- **Horizontal scrollbar** can be enabled if the field is narrower than some images which may be encountered.
- Many word processing documents contain tables or tab-aligned text, so going to the **Table** section of the property inspector and entering a **tabStops** value is recommended.

When you've selected the desired properties for your field, **close** the property inspector.

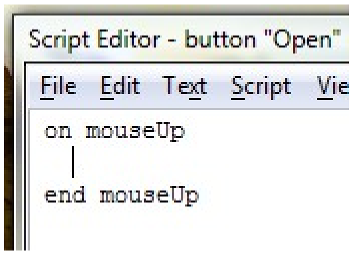
## Selecting a file

Now that you have a field ready to display documents, it's time to find the **file** that we will display. Unless you are bundling documents with your application (which we'll discuss further down), you'll need to let your users select a document.

First the user will need a way of initiating the action, so let's add a **button** to your project. Drag or double-click a button into your stack from the LiveCode **tool palette**, and give the button a **name** or label to display, such as "Open".



Edit its script (*Control-E* or the **Script** button in the LiveCode toolbar) and we'll start adding code to the **mouseUp** handler, which is called when the user clicks on your button.



```
on mouseUp

end mouseUp
```

## The really easy way

For those who don't want to delve much further into technical details at the beginning, I decided to include a way to import files with minimal hassle. It prompts the user for a file, handles basic error checking, and imports any kind of document currently supported by the library.

You just need to provide this command with the name of the field you're using for displaying documents:

```
importDocumentToField "docField"
```

You can specify the (short) name, number, or long name of a field with this command.

Optionally, you can also specify custom prompt text for the file selection dialog the user will see:

```
importDocumentToField "docField","Choose a Microsoft Word or
OpenOffice document:"
```

And now there's also an optional parameter for the file chooser starting folder:

```
importDocumentToField "docField",specialfolderpath("Documents")
```

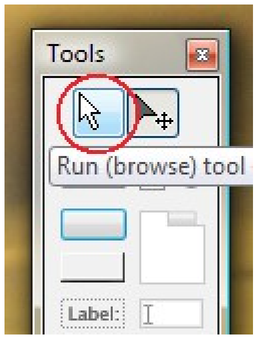
The folder path is used with the file chooser until the user selects a document; then the chooser follows the user's selected folder. Use "#/path" to specify a folder relative to the stack file. The prompt and path parameters can come in either order.

That's it. If all goes well, the selected document will show up in field "docField". If the library recognizes an error that prevents opening the file, the user will be notified.

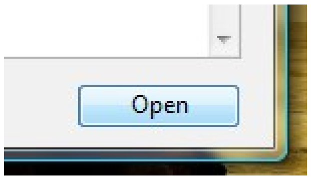
So our button now contains:

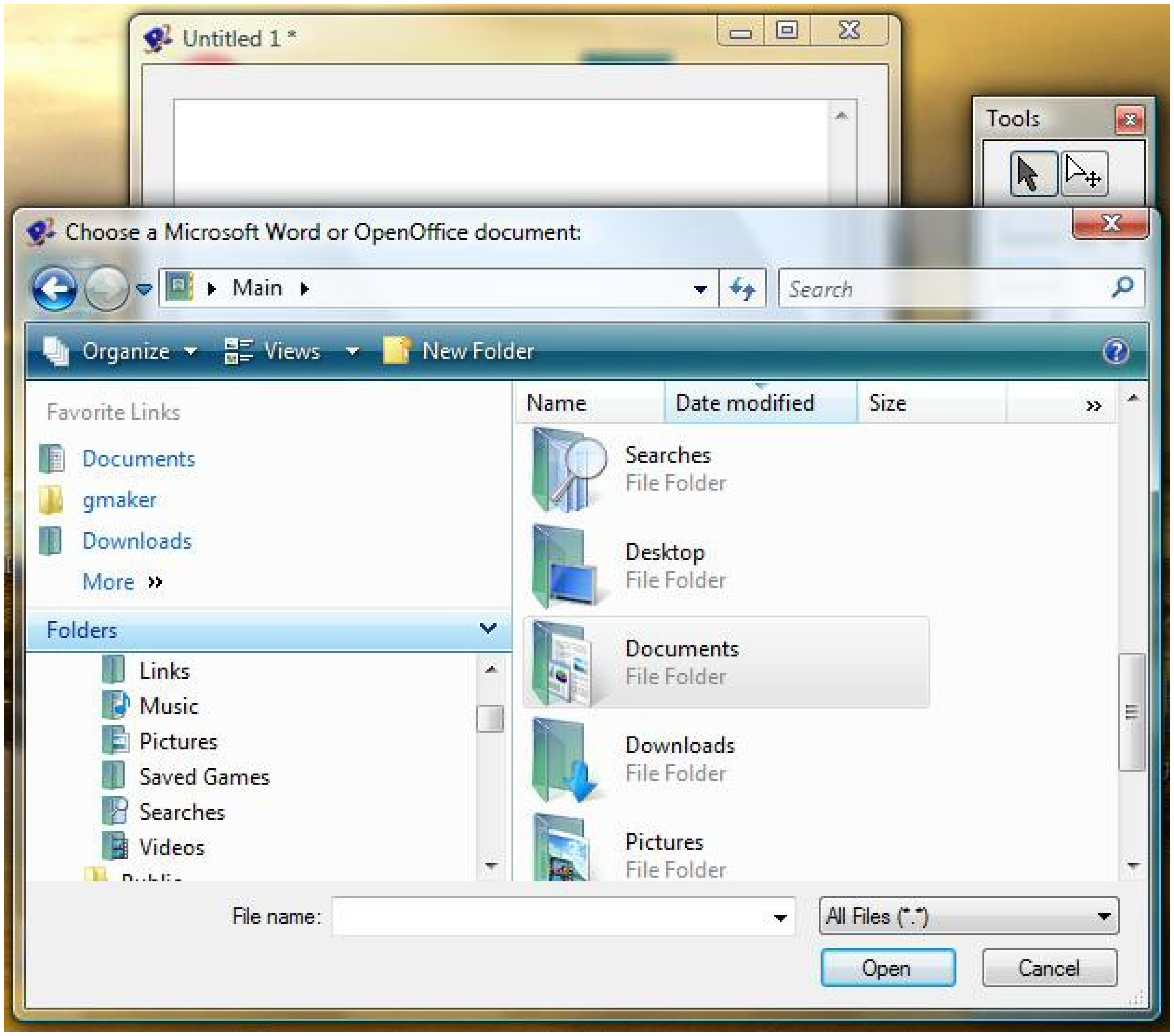
```
on mouseUp
  importDocumentToField "docField"
end mouseUp
```

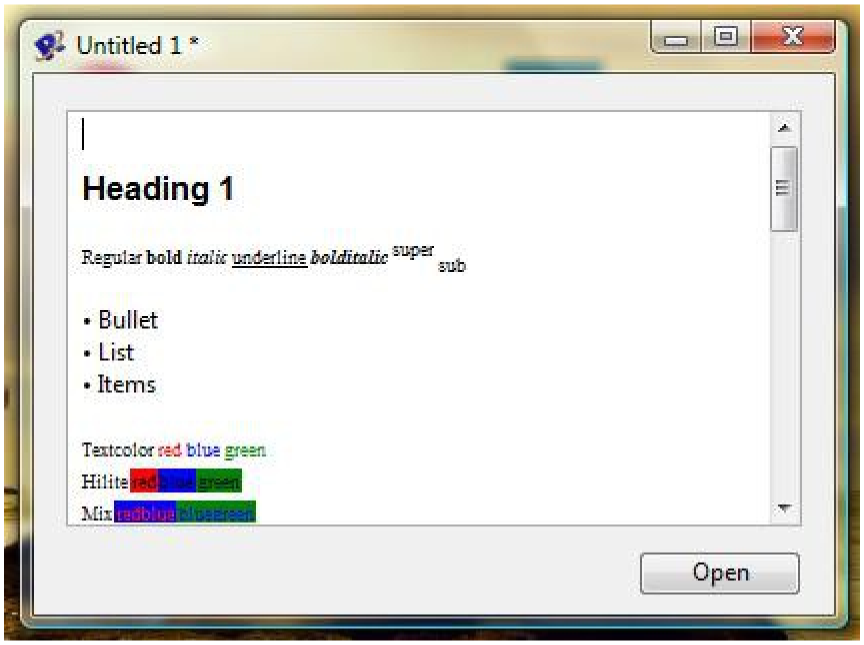
To try out your new document-importing stack, first switch to the **Run/browse tool** in the LiveCode tool palette *(or Control-9)*.



Then click on your **Open** button and find a Word or OpenOffice document to import. You should see the document in your field.


Open


Untitled 1 *
Tools
Choose a Microsoft Word or OpenOffice document:
Main
Search
Organize
Views
New Folder
Favorite Links
Documents
gmaker
Downloads
More
Folders
Links
Music
Pictures
Saved Games
Searches
Videos
Name
Date modified
Size
Searches
File Folder
Desktop
File Folder
Documents
File Folder
Downloads
File Folder
Pictures
File Folder
File name:
All Files (*.*)
Open
Cancel



Heading 1

Regular **bold** *italic* underline ***bolditalic*** $^{super}$ $_{sub}$

- Bullet
- List
- Items

Textcolor red blue green

Hilite red blue green

Mix redblue bluegreen

If that's all you need at this point, you can stop reading today after naming *(Control-K)* and **saving** *(Control-S)* your new stack, and resume your scholarly pursuits in this user guide at some future time when you're further along in your projects.

However, many programmers will want a little more control right away, and they can **read on** to discover more features of the WordLib library.

## The power user method

All right, you decided to roll up your sleeves and do some heavy-duty scripting!

Now we're ready to give the user a **file selection** dialog:

```
answer file "Choose a Microsoft Word or Open Office document:"
```

(You can adjust the prompt text to whatever you like.)

Unless the user cancels, the location of the desired file will be placed in the "**it**" variable. Since other commands later in your script may affect the "it" variable, it's a good idea to put the file location into another variable with an appropriate name:

```
if it is not "cancel" then
  put it into theFile
end if
```

## Importing the file

Now we can really do the import. One of LiveCode's most powerful and straightforward methods of styled text access to fields is via the **htmlText** property, and WordLib is designed specifically for this mechanism. The **htmlText** holds not only the field's text, but also information about fonts, text styles and sizes, and even links and images. (It's very similar to the HTML used in web pages.)

WordLib translates word processing document files into **htmlText** that LiveCode can understand and display in your field:

```
set htmlText of field "docField" to htmlTextOfDocument(theFile)
```

This syntax will open any type of file that WordLib currently handles. (At the moment, that's Word 2007 .**docx** and .**xml**, Word 2003 .**xml**, Open Office .**odt**, and very basic support for the earlier Word .**doc** format.)

Now we have this:

```
answer file "Choose a Microsoft Word or Open Office document:"
if the result is not "cancel" then
  put it into theFile
  set htmlText of field "docField" to htmlTextOfDocument(theFile)
end if
```

(In sections below we'll learn to extend a basic script with error checking.)

## Bundling documents

This part is interesting! When WordLib was born, back in the old days, path handling was a little more work than it is now. Today, loading the library itself or a document relative to your stack is a piece of cake:

```
put specialfolderpath("resources") & "/wordlib.rev" into f
start using f

put specialfolderpath("resources") & "/Docs/help.docx" into f
set htmlText of field "docField" to htmlTextOfDocument(f)
```

Way back then, LiveCode didn't have the “resources” path, so it was harder for LC newbies to handle the paths. I included the **importBundledDocument** command and **wordlibBundled**() function to make things easier.

Now those two handlers may be redundant, but I'll retain the documentation here for now.

Also, this is not to be confused with the “Bundled very tightly” section that comes next! That is a different type of bundling, and is still completely relevant, providing capabilities not duplicated by LC.

WordLib isn't limited to opening files that the end-user has created. How about **bundled** documentation that you've written and want to pass on to your users? (Just like this user guide you are reading.) Composing documentation in your regular word processor gives you lots of power and convenience in designing your document, and with WordLib, you can display it right inside your software.

For example, you could write a handler like this:

```
on showHelp
  set itemdel to "/" -- file path is divided by slashes
  put the fileName of this stack into f -- location of current stack
  put "help.docx" into last item of f -- substitute document name
  set htmlText of field "docField" to htmlTextOfDocument(f)
  set itemdel to comma -- return setting to default value
end showHelp
```

But that's a bit long, isn't it? To make it more convenient, WordLib includes the **wordlibBundled**() function to get the full file path of a bundled document. That makes your scripts shorter and easier to read:

```
on showHelp
  put wordlibBundled("help.docx") into f -- document location
  set htmlText of field "docField" to htmlTextOfDocument(f)
end showHelp
```

That's better, but it's still two lines of code, and it also lacks error checking. WordLib makes it even easier with the **importBundledDocument** command. Just give it a file name and field name/number, and the handler does the rest, automatically reporting any errors to the user:

```
on showHelp
  importBundledDocument "help.docx","docField"
end showHelp
```

If there is a recognized error when opening a bundled document (for example, if files were moved around on the user's computer and the document can't be found) the user will be **informed** without further effort on your part.

On the other hand, **advanced** users may want to take a back-up action or display a custom message rather than the default error dialog.

In this case, you can use the silent option by passing "**silent**" or "**true**" as the third parameter so that **importBundledDocument** will not display errors. Then (and this is true regardless of whether you are using the silent option) you can check the **result** for any error message.

So we could end up with something like this:

```
on showHelp
  importBundledDocument "help.docx","docField","silent"
  if the result is not "" then -- error, show backup text
    put the myBackupText of this stack into field "docField"
  end if
end showHelp
```

This assumes that the bundled document file is in the same folder as your project stack. If you'd rather put bundled documents in a separate folder for greater organization, you can modify the path as needed. For example, if your project will be built into the standalone executable file and bundled documents are in a folder called "Docs":

```
put wordlibBundled("Docs/help.docx") into theFile
```

Or if your main project stack file is in a "Data" folder rather than built into the standalone:

```
importBundledDocument "../Docs/help.docx","docField"
```

## Bundled very tightly

Bundling documents as separate files distributed with your application has several advantages. It **conserves** memory and stack size. You are also providing materials that can be viewed in an **external** viewer or word processor as well as inside your application.

However, there are also some unique benefits to **storing** documents right in your stack. This will **eliminate** any chance that document files could be accidentally moved around by the user. It also ensures that a help file can be displayed if your stack can be run, even if your application was not installed correctly or was run from a compressed archive which the user still hasn't expanded.

In addition, documents will be viewed **instantly** without any need for loading, just like any other regular field text. Your word processor can become an external editor for composing and styling text to use in LiveCode fields!

Obviously, whether to bundle documents as separate files or built into stacks could be a big decision, because both methods have excellent advantages. You can even do both, providing separate documents to save memory and allow external viewing, but also including a backup text just in case. (The **Try WordLib** stack takes this approach.)

But whatever you decide, WordLib will provide the means for you to bundle your documents, including documents stored inside your stacks.

In its most basic form, this can consist of simply **importing** a document into a field (using one or more of the commands and functions discussed in previous sections) and then **saving** your stack. If your field only needs to display that particular document, and the document contains no **images**, you're all done!

(But please remember to **register** WordLib if you use it to import and save bundled documents for any commercial or non-personal use, including freeware, shareware, use at work, consulting, use in the classroom, e-books, etc., or if you use it regularly and find it useful. Thanks for your support!)

Now to consider something a little more complex: what if one field needs the ability to display multiple documents? This is generally a matter of storing and recalling the **htmlText** property of a field. You can use WordLib to import a document, then **save** the document into your stack:

```
set the myHelp of this stack to the htmlText of field "docField"
```

and later recall it when needed:

```
set the htmlText of field "docField" to the myHelp of this stack
```

Viola! Import other documents and save to other properties, and you can switch between as many built-in texts as you like.

However, there's just one catch: if a document contains **images**, these will be located in files elsewhere on your computer; they are not stored directly in the **htmlText** property. (Image file storage is discussed in the Images section below.) This means that storing an imported document with images in your stack would require some extra work.

Fortunately, WordLib handles this for you.

## Bundling the images too

Using the **saveBundledDocument** command, you can store the contents of an imported document along with any associated **images**, which will be imported into your stack:

```
saveBundledDocument "docField","myHelp"
```

This will store the styled text in field "**docField**" in stack custom property **myHelp**. Any images will be placed invisibly on the current card of your stack. Of course, you can use any valid field name or property name instead of "**docField**" and "**myHelp**".

(The **saveBundledDocument** command is only available to registered users.)

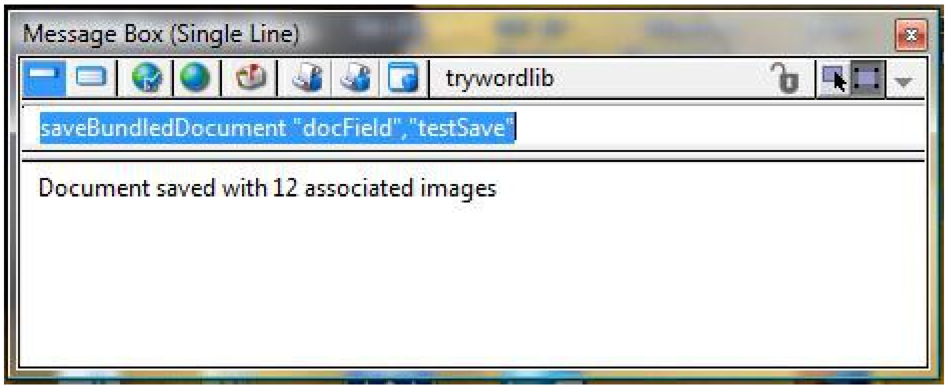
For example, saving a document containing three images with the above command would result in a stack property named "**myHelp**" containing the document text, and your image names might be:

- wordlib-myHelp-0001.jpg
- wordlib-myHelp-0002.gif
- wordlib-myHelp-0003.png

These images will be hidden, so you won't see them appear on your card unless you choose to show hidden objects.

You need to already have a document imported in a field, using a method discussed in previous sections, before using the **saveBundledDocument** command; it doesn't accept a document file path as a parameter. The field contents will appear the same after executing the command, but the field's **htmlText** will be updated to use the newly imported stack images rather than external image files.

WordLib will return information in **the result** to let you know if the save was successful and how many images were imported. (The feedback is actually pretty nifty, try it!) If you are using the message box to issue your commands, you'll see something like this:

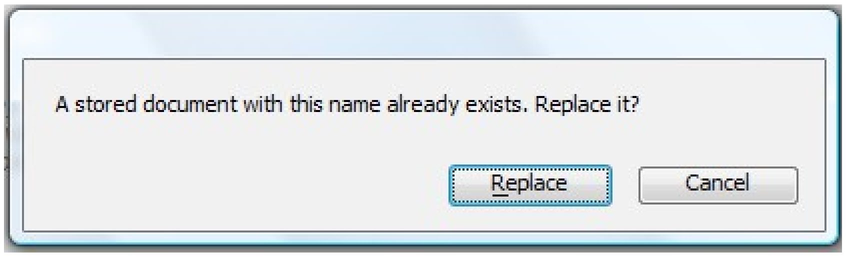


```
saveBundledDocument "docField","testSave"
```

```
Document saved with 12 associated images
```

Saved documents take up some space in your stack (especially images) so be sure to tidy up if you save a bundled document and then decide not to use it. Fear not, WordLib offers a couple of additional features to make **maintenance** easy.

If you use the **saveBundledDocument** command with a stack property name that already contains a document, WordLib will confirm whether you want to replace the previous contents. If so, it will automatically delete any images on the current card that are associated with the old contents before saving the new document.



A stored document with this name already exists. Replace it?

This is handy for updating a document by importing again after editing in your word processor, or replacing it with a completely different document. The feature is built-in, so you don't need to take any special action except for confirming whether to replace or cancel.

How about deleting without replacing? You can delete a bundled document and any associated images using the **deleteBundledDocument** command with the stack property name as a parameter:

```
deleteBundledDocument "myHelp"
```

This will delete the stack property myHelp and any images on the current card with names containing "**wordlib-myHelp-**". WordLib will confirm your intent to delete the document before taking action; if you want to skip the confirmation, use "silent" or "true" as the second parameter:

```
deleteBundledDocument "myHelp","silent"
```

Again, WordLib will provide feedback in **the result** on what was deleted as well as what was saved.

Since images are saved on the current card in your stack, be sure to use these commands while on the

**same card** if you are updating or deleting a document.

(If you try to update or delete a document with images and WordLib reports 0 images deleted, you'll know that something is amiss—for example, you are on the wrong card, you have already deleted the document, or the stack may be protected. WordLib will always try to delete both the stack property and any images, so if you issue a command while on the wrong card, you can go to the correct card and try again.)

If you prefer, you can maintain the parts of your stored documents manually by editing stack properties *(Control-K, Custom Properties section)* and by viewing hidden objects *(View>Show Invisible Objects)* or using the **Application Browser** *(Tools>Application Browser)* to manage images.

## Being more specific

The **htmlTextOfDocument** function opens anything WordLib can handle. It's simple, and it also means that if WordLib supports other formats in the future, your projects **automatically** handle them without changes.

However, if you're after a particular type of file, there are also more specific versions of the import function. These currently include:

```
get htmlTextOfDocX(theFile) -- MS Word .docx, .docm, .xml files
get htmlTextOfODT(theFile) -- Open Office .odt files
get htmlTextOfDoc(theFile) -- MS Word .doc files
```

## Checking the file type

If you're making software for other people, you will probably want to check that the file opened is a supported type.

(The **importDocumentToField** command already includes this feature, so further checking is not necessary unless you want to provide a custom message or action.)

The **wordlibSupports** function determines whether a file type is supported:

```
if wordlibSupports(theFile) then
  set htmlText of field "docField" to htmlTextOfDocument(theFile)
end if
```

This function looks at the file name extension and determines whether it is a type currently supported by WordLib. You can do the same by checking the extension yourself, but the function offers the advantage of not needing to update your code whenever WordLib is updated to support additional file formats or to utilize more advanced methods of differentiating files.

These **supported** file formats are also contained in the result of this function:

```
get wordlibFileTypes() -- supported extensions, space-delimited
```

The **htmlTextOfDocument** function and other import functions will also return an error code (see next section for error format) if an unsupported document is presented and can't be opened.

Another option would be to use file type filtering in the file selection dialog to help ensure that only correct files are chosen—see LiveCode documentation about "**answer file**" for details.

## Error checking

If you want to provide users with customized information when things go wrong, or have your software take its own actions based on the situation, you can use the error codes and messages that WordLib provides.

Major errors will prevent WordLib from importing a document. These will be returned directly as the result of the **htmltextOfDocument** function, or automatically displayed in answer dialogs when using the **importDocumentToField** command.

The first (comma-delimited) item of the error result will be the word "**error**", and the second item will be a one-word **error code**. You can use this to respond to an error with your own action. (See the file **errorcodes.txt** for a list of codes.) The remainder of the error result (item 3 to -1) will be a default message and any other information about the error, which can be passed on to your users if you wish.

Here's how error checking in a script might look:

```
get htmltextOfDocument(theFile) -- import document

if item 1 of it is "error" then
  if item 2 of it is "cantOpenDoc" then
    -- provide your own handling for this error
    show image "bad-document"
    answer "Sorry, this file appears to be damaged or invalid."
  else -- other errors
    answer item 3 to -1 of it -- show error message
  end if
else -- no major error
  set htmltext of field 1 to it -- display document
end if
```

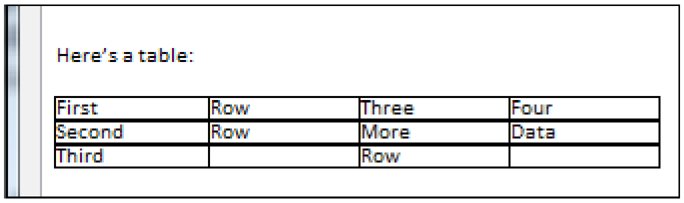
Sometimes there are errors or warnings which do not prevent document import but may affect some contents. These minor errors are placed in the **gwordlibErrors** variable. Each line contains one error.

The anatomy of each minor error line is similar to that of the major error result: the first item is an error code, and the rest (item 2 to -1) is a default message along with any other information about the error.

## Tables

**Tables** in documents will be displayed **inline** in your field, with the data in columns and lines separating cells.

WordLib **pioneered** the display of tables inside fields in LiveCode before the IDE officially supported this capability! It worked very well for simple tables. However, some types of Unicode and styled text could disrupt the display, so now WordLib 2.0 utilizes the new LiveCode build-in table support to display MS Word tables.



Here's a table:

| | | | |
|---|---|---|---|
| First | Row | Three | Four |
| Second | Row | More | Data |
| Third | | Row | |

**WordLib 1.5**: If table columns are squeezed too tightly to show data, you can adjust your field's **tabStops** property. (Having a **tabStops** setting is strongly recommended for any field that will import documents.) For really advanced use, you could program an interface to let the user interactively adjust tabstops for a field.

**WordLib 2.0**: The import will **follow** the document's specifications for each imported table and set column widths accordingly, just as you would expect.

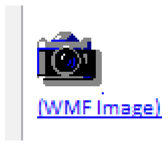
If a table contains a large amount of data per cell, or multiple lines of text inside a cell, it may not display as nicely in your LiveCode field.

## Images

Word processing documents often have embedded **images**. WordLib will display them whenever possible (images that LiveCode can display in fields, such as **GIF**, **JPG**, and **PNG**) inline in your document, similar to what you will see in your word processing software.

(For document file formats with limited support—currently MS Word legacy **.doc** files—usually images will not be shown.)

For basic use of the library, that's all you need to know—just import documents and all supported images will appear. For advanced users, here is additional information.



Any unsupported images (which LiveCode can't display in fields) will be marked with a placeholder. You will see a **camera icon** with a link indicating the image file type. If you click on the camera icon or link text, you can open the image for viewing with other software. (Assuming that your computer has appropriate viewing software and file associations.)

See the **Try WordLib** stack's field script to learn how to handle the link clicks in your own stacks.

Remember that if a field is not locked, the user must hold down the **Control key** while clicking.

If you'd rather show a different icon or cut out the unsupported image links altogether, try one of these user options:

```
-- turn off image links/icons
global gwordlibImageLinks
put false into gwordlibImageLinks

-- provide name/id of image in your stack
global gwordlibImageLinkIcon
put myIcon into gwordlibImageLinkIcon
```

All images embedded in documents are first exported to a temporary folder called the **WordLib Media Folder**. The default location of this folder on your computer varies per platform and can be found by checking the value of the global variable **gwordlibMediaFolder** after importing a document. If you need images from a file, open this location in your computer, and there they are!

(For convenience, the command **openWordLibMediaFolder** will attempt to open this folder on your computer.)

Images in the **Media Folder** are named by document number (four digits), hyphen, image number (four digits), dot, and file extension. Example: 0001-0001.jpg. This allows multiple imported documents to reside in your projects without stepping on each other's toes. (Up to 9999 documents per session with 9999 images per document.)

The **Media Folder** is cleaned out (all files deleted) when the WordLib library is loaded, so the files in the **Media Folder** last for one session of your application. If you would like to clean the **Media Folder** on command, call the **cleanWordLibMediaFolder** handler. If you don't want the **Media Folder** cleaned out automatically, set the global variable **gwordlibAutoCleanup** to false.

If you want to work with embedded images in your project scripts, you can get and set the value of **gwordlibMediaFolder** as needed. The value this variable has before an import will be the folder path used. (This is an absolute path, and it must contain “WordlibMedia” if you wish to empty it with **cleanWordLibMediaFolder** or **gwordlibAutoCleanup.)**

## Text size

One "gotcha" of the **htmlText** system in LiveCode is that **<font size="12">** indicates actual text font size of 12, while **<font size="3">** indicates an html-style size which also gives you text with an actual point size of 12, rather than 3.

(This is true whenever you use the **htmlText** property; it is a feature of LiveCode and is not caused by WordLib. It is helpful in some situations, but causes side effects in others.)

Another example: **<font size="7">** produces 25-point text.

LiveCode interprets any **htmlText** font size number from 1 to 7 as an html-style size, and anything larger is used as real point size. Thus, the smallest text size you can set with **htmlText** is 8 points.

To minimize the side effects of this system (such as 5-point text displaying twice as large as 8-point text) currently WordLib will substitute size 8 text for anything smaller when importing a document.

## Recap

For a handy list of all WordLib's commands, functions, and user options, see the document **WordLib Quick Reference.odt**. (You can view the reference document in OpenOffice, or in LiveCode by opening the **Try WordLib** stack and using the **View Help** button.)

## After Registration

If you haven't **purchased** WordLib yet, just click on the **Purchase** button in the **Try WordLib** stack and choose between the **CurryK** and **LiveCode** stores to process your payment.

You'll receive a License key. TOS: Keep this key confidential, and password-protect any stack containing it before distribution!

You can use the **registerWordLib** command with your key to access all features. This should be placed in your project script, right **after loading** the library.

```
on preOpenCard
  -- load the embedded library
  start using "wordlib"
  registerwordlib "your code here"
end preOpenCard
```

My products do not embed themselves deeply in your system or insert code into your scripts; this method lets you remain entirely **in control** of your LiveCode stacks and computer at all times, with no install/uninstall headaches! You can also enter the **registerWordLib** command in the **LiveCode Message Box** for testing or single-session use.

Thanks for your registration!

## Writing home

If you have comments, feel free to drop me a line.
I'd love to hear about how people are using WordLib!

Run into a problem? First read the FAQ! It'll be a growing resource.

If your issue is not listed, be sure to mention your OS (such as Win 10 or Mac Big Sur), your LiveCode version (such as 9.6.5), and a reliable recipe or document file attachment. If you are using a standalone, mention that too.

FAQ-level support is free. Other support is available at an hourly rate.

info@curryk.com

However, it's also perfectly okay to write when there is no problem. :)

## Conclusion

I hope you enjoy **WordLib**. Be sure to let me know how you're doing. Thanks for your support!

Best wishes,

**Curry Kenworthy**

**E-mail:** info@curryk.com

**Web**: http://livecodeaddons.com/wordlib.html