DigHT 310
Using External Files

Objectives

By the end of this reading you should understand the following:

1. The two approaches to accessing external files on the local file system.
2. Know how to work with files using the open file-close file model.
3. Understand that there are different "modes" of the open file command, depending on what you plan to do with the file after you open it.
4. Recognize that good programming practice demands that you always close a previously opened file when you are finished accessing it.
5. Know how to work with files using the resource-oriented (URL) model.
6. Understand how file permissions can limit what you can do with files, regardless of the access model you use.

Often you may want to use data stored in external files in your LiveCode program. LiveCode has a rich set of tools that make reading, writing and modifying external files simple. This lesson will look at the basic tools and techniques for working with external files.

Two Models for working with external files

There are two different approaches for reading and writing external files. One model is oriented toward local, disk-based files systems. The other is oriented toward a world connected by the internet.

Model One: File-oriented—a little more work for greater flexibility

Note: This model is explained in greater detail in the LiveCode stack ExtFilesLect.rev. Remember that you can launch any stack stored on a web server by using a statement like this:

```
go to stack URL "http://dight210.byu.edu/Resources/InClass_Work/ExtFilesLect.rev"
```

Do the activity "Exploring the open and close commands".

Under this model, before you do anything with a file it must be accessed, or formally "opened". The `open file` and `close file` commands perform the "housekeeping" functions that are necessary to adhere to the rules of the computer's file system, freeing the developer from that task.

By default the `open file` command looks for the file to create the file (if it doesn't already exist) in the same folder as the LiveCode application. If another location is desired, the entire pathname must be specified. LiveCode uses Unix file path conventions, so the slash (/) character is used to delimit levels in the file path. Users of the Classic Mac OS therefore must get used to using a slash character instead of a colon to delimit levels, while Windows users must get used to using a slash rather than a backslash (\).

```
open file "myFile.txt"
close file "myFile.txt"
```

-- will open or create a file in the LiveCode application folder by default

Use this format on Unix and Macintosh systems. Note the / at the beginning of the file path:

```
open file "/MyHD/Documents/MyProject/myFile.txt"
close file "/MyHD/Documents/MyProject/myFile.txt"
```

On Windows systems you specify the Drive letter:

```
open file "C:/Documents/MyProject/myFile.txt"
close file "C:/Documents/MyProject/myFile.txt"
```

Beware of the common errors when using `open file` and `close file`:

- Giving an incomplete or incorrect path name;
- Typing the file name differently in the `open` and `close` commands;
- Trying to open an already opened file;
- Trying to close an already closed file;
- Forgetting the key word `file`.

Variants of the open file command

Aside from the basic forms, there are several variants of this command that create specific conditions when the file is opened:

`open file for update` - (Default form) opened file will be available for both reading and writing.

`open file for read` - Opened file will be available for read only. You must use this form for opening files from read-only media (like CD-ROMs); if you try to use any other form the subsequent read access will fail.

`open file for write` - Next write to file command will completely replace the file's contents.

`open file for append` - Next write to file command will add its data to the end of the file without replacing current contents.

Reading the contents of files

Once a file is open, you often will want to get its contents so you can use it in your program in some way. Use the `read from file` command for this. Here is a common usage of this command:

```
read from file filename until end
```

Where *filename* is a properly-formed, full path name of the file you want to read from. Remember, if you don't use a full path designation LiveCode will expect the file to be in the `defaultFolder`. The following sequence will read all of the data in `myFile.txt` and put it into the `it` variable:

```
open file "/My HD/MyFolder/myFile.txt" for update
read from file "/My HD/MyFolder/myFile.txt" until end
close file "/My HD/MyFolder/myFile.txt"
```

Variants of the read from file command

`read from file ... until end|empty|EOF` -- these are synonyms. All three will read until reaching the end of the file.

`read from file ... until string` -- where *string* is any text string. Will read until encountering the string in the file.

`read from file ... at startPoint for amount` -- where *startPoint* and *amount* are positive integers. Will read starting at startPoint for amount characters.

`read from file ... for amount words|items|lines` -- where *amount* is a positive integer. Reads the specified number of chunks.

Common errors

- No `until` or `for` parameter provided.
- Filename is different from opened file.
- File not previously opened.
- Wrong format of `open file` command. For example, the following forms will cause an error when followed by a read command:

```
open file "myFile.txt" for write
open file "myFile.txt" for append
```

Writing data to files

The other common operation you will want to do with files is to write data to them. For these operations there is the `write to file` command. Here is a typical usage of this command:

```
write "Hello World!" to file filename
```

Where *filename* is a properly-formed, full path name of the file you want to read from. Remember, if you don't use a full path designation LiveCode will expect the file to be in the `defaultFolder`. The following sequence will overwrite the first five characters of `myFile.txt`:

```
open file "/MyHD/MyFolder/myFile.txt"
write "Hello" to file "/MyHD/MyFolder/myFile.txt"
close file "/MyHD/MyFolder/myFile.txt"
```

The write command must be preceded by either the `open file for update`, `open file for write`, or `open file for append` form of the `open file` command. Preceding a `write` command with the `open file for read` form, for example, will cause the `write` command to fail, and the `result` function will return "File is not open for write."

By default, writing starts where the last `read` or `write` command left off, assuming the file was not closed since the previous `read` or `write`.

Variants of the write to file command

`write...to file...at start` -- where *start* is a positive or negative integer. For a positive number the write begins `start` characters after the beginning of the file; for a negative number the write begins `start` characters before the end of the file.

If no *start* is specified, writing begins at the place the last read or write operation left off, or at the first character if this is the first read or write since the file was opened.

`write...to file...at EOF|end` -- Starts writing after the last char in the file.

Common errors

- Overwriting an existing file with new file of the same name.
- Using a filename that is different from opened file.
- Trying to write to a file not previously opened.
- Using the wrong format of open file command

```
-- All of these can be followed by a write to file command:
open file "myFile.txt"
open file "myFile.txt" for update
open file "myFile.txt" for write
open file "myFile.txt" for append

-- This form will cause an error when followed by wrote to file:
open file "myFile.txt" for read
```

Model two: Resource-oriented—simple and far-reaching

The mid-1990s was a watershed period for personal computing. During this time the World Wide Web revolution started a paradigm shift in the way people worked with their computers. No longer were users limited to accessing only data on their local hard drives. Now, through the medium of a web browser they could access information on a huge, interconnected "web" of computers around the world. The internet became a huge presence almost overnight.

The inventors of the Web needed a simple way to reliably retrieve different types of data on the literally millions of computers on the internet. They came up with the concept of the Uniform Resource Locator, or URL. This is most familiar to users of the Web as the "web site address" they enter into their browser's address field to go to a new web site—familiar addresses like www.ibm.com, www.byu.edu, www.nasa.gov, livecode.com, etc. These URLs often include file path and file name information in addition to the basic web site address. For example, an address like livecode.byu.edu/dight310/schedule.html tells us that the resource we want is on the web server **livecode.byu.edu**, in a folder named **"dight310"** and in a file named **"schedule.html"**.

What many Web users don't realize is that URLs can locate much more than "pages" on the internet. There are a number of URL **schemes** that specify the *method* or *protocol* for retrieving the resource you are trying to get. You see this in the **http://** portion of the URL, which web browsers automatically provide if you do not type it before the address. HTTP, or Hypertext Transfer Protocol, is far and away the most commonly-used scheme on the internet. But you've probably noticed others, like **https**, Hypertext Transfer Protocol Secure; **mailto**, Email address; or **ftp**, File Transfer Protocol. There is even a scheme, called **file**, for accessing files on the local file system.

Another thing many Web users don't realize is that any computer application, not just web browsers, can be written to access internet resources. Some common examples include email applications, like Microsoft's Outlook and Apple's Mac OS X Mail; iTunes, which connects directly to Apple's online music store; and the many applications that include a "Check for Updates" feature.

LiveCode was written to allow simple access to internet resources using URL-style addresses and schemes. What this means to you as a developer is that you can easily access virtually any public data resource on any web server anywhere in the world. LiveCode supports the following URL schemes:

http: Hyper-Text Transfer Protocol—a page on a web server
https: Hyper-Text Transfer Protocol Secure—same as http, but for accessing a page on a secure web server
file: a text file on the local disk (not on a server)
binfile: a binary data file on the local disk. Use the **binfile** scheme to read in media files, such as an audio or image files, or files saved in text encodings that are different from your computer's native encoding, such as UTF-8.
ftp: File Transfer Protocol—a directory or file on an FTP server
resfile: on Mac OS and OS X systems, the resource fork of a file; rarely used nowadays

The URL keyword—LiveCode's key to accessing external resources

In the context of the present discussion about working with local files, this means that you can read a file on your computer's hard drive in a single command by using the **URL** keyword:

```
put URL "file:myFile.txt" into field "myField"
```

As you would anticipate from past discussions, this will place the contents of the file "myFile.txt", located in the current **defaultFolder** into field "myfield". You could access any file on your hard drive, or on any mounted volume, simply by supplying the full file path to that file. (See the earlier lesson for a review of how to refer to external files for a review of file path formats.)

You are not limited only to retrieving entire files. A URL can be considered a full-fledged LiveCode container, just like a field or variable. That means you can access chunks of data in a file using lines, characters, words, items, etc.

```
put line 1 of URL "file:myFile.txt" into field "myField"
put word 6 to 10 of URL "file:myFile.txt" into myVar
get item 5 of URL "file:/Hard Drive/My Folder/myFile.txt" -- Mac format filepath
get item 5 of URL "file:C:/Documents and Settings/localuser/My Documents/myFile.txt"  -- Windows format filepath
```

Nor are you limited only to *reading* from URL resource files. You can write data to any URL container in the same way to put data into a field or variable:

```
put field "myField" into URL "file:myFile.txt"
put return & line 1 of field "userName" after URL "file:userList.txt"
put it into item 5 of URL "file:myFile.txt"
put "Text" into word 3 of URL "file:/Users/Documents/Test.txt"
```

If you put data into a URL container that doesn't exist, LiveCode will create the file, then write to it. However, if you try to put data into a URL container in a folder that doesn't exist, the command will fail. One technique to ensure you don't try to write to a non-existent folder is to check to see if it exists first:

```
# in this example the folder would be in the defaultFolder
if there is a folder "myFolder" then
   put field "myfield" into URL "file:myFolder/myFile.txt"
end if
```

Of course, the ability to put data into a URL assumes that the file system will allow you to do so. For security reasons, many areas of a file system may only be writable by certain users with high levels of access to the hard drive. Additionally, some disks, such as CD-ROMs or DVD-ROMs, are not writable (ROM stands for "Read-only Memory") by nature. If you try write to a URL that is not writable, the command will fail silently (with no warning message.) To check to see whether a `put into URL` command was successful, you should check **the result** function immediately after issuing the command, like this:

```
put field "myfield" into URL "file:/System/myFile.txt"
# (mere mortals can't write to the Mac System folder!)
put the result into tResult
if tResult is not empty then
   answer "Your operation failed with this error: " & tResult
end if
```

So far we have only discussed the **file** protocol for URLs. There are several others that LiveCode supports. Most of these we will cover in a later discussion. But let's look briefly at the **binfile** protocol. The binfile protocol is also used to access files on the local file system. The main difference between the **file** and **binfile** is that the **file** protocol is intended for files containing platform-native ASCII text. The **binfile** protocol is for accessing files containing binary data—things like Unicode text, images or audio files, for example. When you use the **file** protocol to retrieve the contents of a file, it modifies the line ending characters to match the current operating system. If you don't know what this means, don't worry—we'll cover this in a future topic. For now it's sufficient to understand that you should use **file** for plain, platform-native ASCII text files, and **binfile** for everything else.

On file location and the default folder

Accessing common storage locations

Absolute paths have several inherent limitations. Most notably, if you create a program on one machine, the absolute paths will most likely be different on another one (due to different folder names, different user logins, different platforms, and so forth). But LiveCode provides tools for adapting absolute file paths to the current host system. Most common operating systems have standard folders for storing certain types of information. For example, preference files tend to be stored in a specific place; each user has a "home" folder where they can save their files; and there are typical places for storing temporary files. As a cross-platform development tool, LiveCode gives you an easy way to find out where these folders are located on the current host operating system, a function called `specialFolderPath()`.

How does this function work? Let's say, for instance, that you want to save a file to the user's Desktop. Simply use the `specialFolderPath()` function with the argument "desktop":

```
put specialFolderPath("desktop") into tFolderPath
put tFolderPath & "/mydocument.txt" into tFilePath
put field "importantStuff" into URL ("file:" & tFilePath)
```

The `specialFolderPath()` function returns a file path to the folder appropriate for the current operating system. For example, on Mac OS X `specialFolderPath("desktop")` might return something like "/Users/myname/Desktop". On Windows it might be something like "C:/Users/myname/Desktop".

There are several common folder identifiers you can plug in to this function, including "home", "preferences", "temporary", and "fonts". Each identifier returns the platform-specific path to the given folder. See the LiveCode dictionary entry for specialFolderPath for a complete list.

One particularly useful folder identifier is "resources". When you call `specialFolderPath("resources")` you get the file path to the folder your stack is saved in. That makes it simple to build file paths to media assets that are stored in the same folder as your stack.

A note about special folder paths on mobile platforms. The `specialFolderPath()` function works on iOS and Android, but because the file system access is much different on mobile versus desktop platforms, the available folder identifiers for mobile platforms are different, too. Perhaps the most useful folder identifiers for mobile devices are `engine`, `documents`, and `cache`. The `resources` identifier is also useful in the mobile environment; it is a synonym for `engine`.

- `specialFolderPath("engine")` -- returns the file path to the folder where your app is located in the mobile device's file system. This is most often where you would store media file resources like image and audio files for access by your app.
- `specialFolderPath("documents")` -- returns the file path to the folder where your app is permitted to save permanent documents.
- `specialFolderPath("cache")` -- returns the file path to the folder where your app can save temporary files.

Relative File Paths and the defaultFolder Property

What if you want to access files that are in a folder other than one of the special folders accessible through `specialFolderPath()`? LiveCode provides a way for you to specify where files should be created by default and where LiveCode should look for files when no file path is specified. You can do this with the **defaultFolder** property. The `defaultFolder` is a property that applies to the entire LiveCode environment. Such properties are called *global* properties. If you examine the `defaultFolder` property immediately after launching LiveCode:

```
put the defaultFolder into message box
```

You will get a result something like this (for Mac/Linux):

```
/Applications/
```

Or this (for Windows):

```
C:/Program Files/RunRev/LiveCode
```

You can change the `defaultFolder` by setting it to the desired folder:

```
set the defaultFolder to "/MyHD/MyProject/textfiles"
```

Once the defaultFolder is set, any reference to a relative file path will begin at the specified default folder to derive the full file path. Setting the defaultFolder persists only until LiveCode is shut down, so if you want this folder to be used every time you open your stack, you should probably set the `defaultFolder` in an `openStack` or `openCard` handler.

Here is how to set the `defaultFolder` to the folder where the currently open stack is saved:

```
set the defaultFolder to specialFolderPath("resources")
```

In versions of LiveCode before v. 6.7, the `specialFolderPath("resources")` option did not exist, so in older versions you would have to do it like this:

```
get the effective fileName of this stack
set the itemDelimiter to "/"
delete last item of it
set the defaultFolder to it
```

Ways of choosing files

Several commands are provided that make it easy to choose a location for the file. They can be used in conjunction with the `get/put URL` method, or with `open`, `read`, `write`, and `close` commands. These commands are intended to present familiar open and save style dialog boxes, consistent with the operating system you are using.

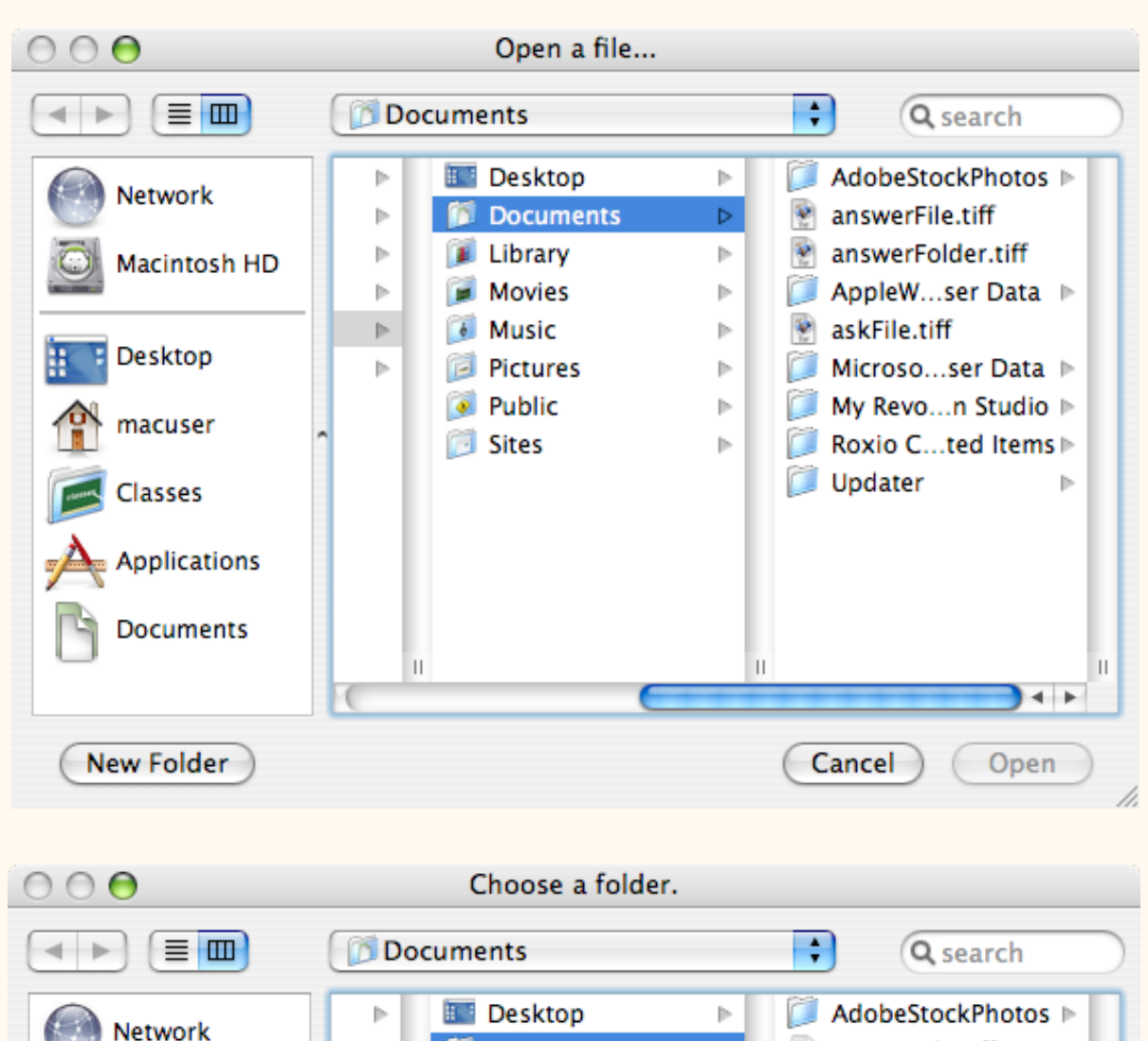
answer file

syntax: `answer file prompt` where *prompt* is a text prompt that will appear in the dialog.

The `answer file` command brings up a standard file selection dialog, such as you see when choosing Open... from the File menu in most applications. When the user select the file, its full path name is returned to the special variable `it`. Keep in mind that `answer file` does not actually open the file, but only returns the name and path of the selected file. The actual opening and reading have to be scripted.

Here is a sample script segment for choosing and opening a file using `answer file`:

```
answer file "Open a file..."
put it into tFilePath
put url ("file:" & tFilePath) into fld "fileContents"
```



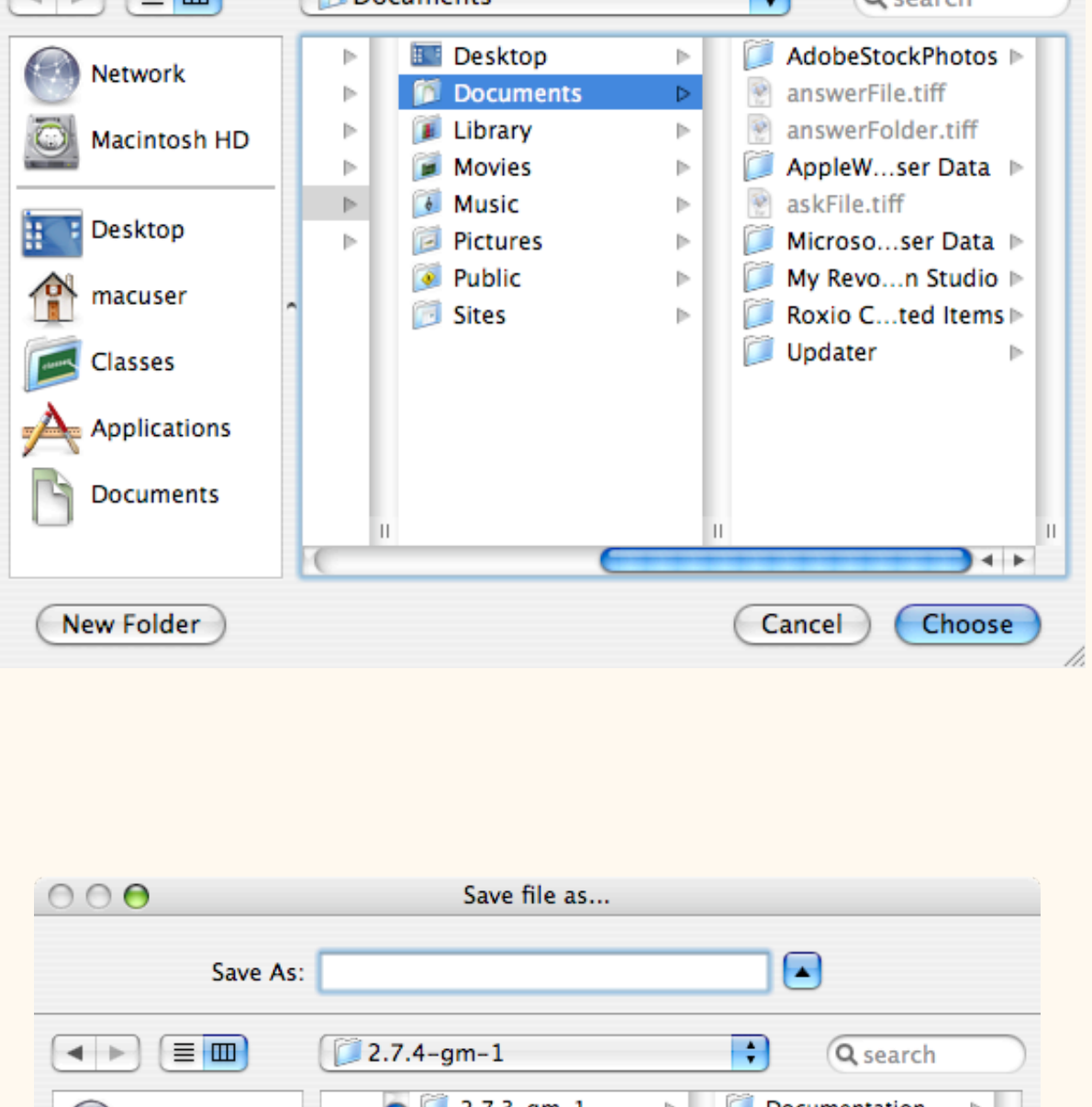
answer folder

syntax: `answer folder [prompt]` where *prompt* is an optional text prompt that will appear in the dialog.

The `answer folder` command brings up a standard Open dialog to let the user select a folder. The full pathname of the selected folder is returned in the `it` variable. This command can be used to let the user choose where a file is to be saved, or where to look for an existing file.

Here is a sample script segment for allowing a user to select a default folder:

```
answer folder "Where are your audio files located?"
put it into tFolderPath
set the defaultFolder to tFolderPath
```



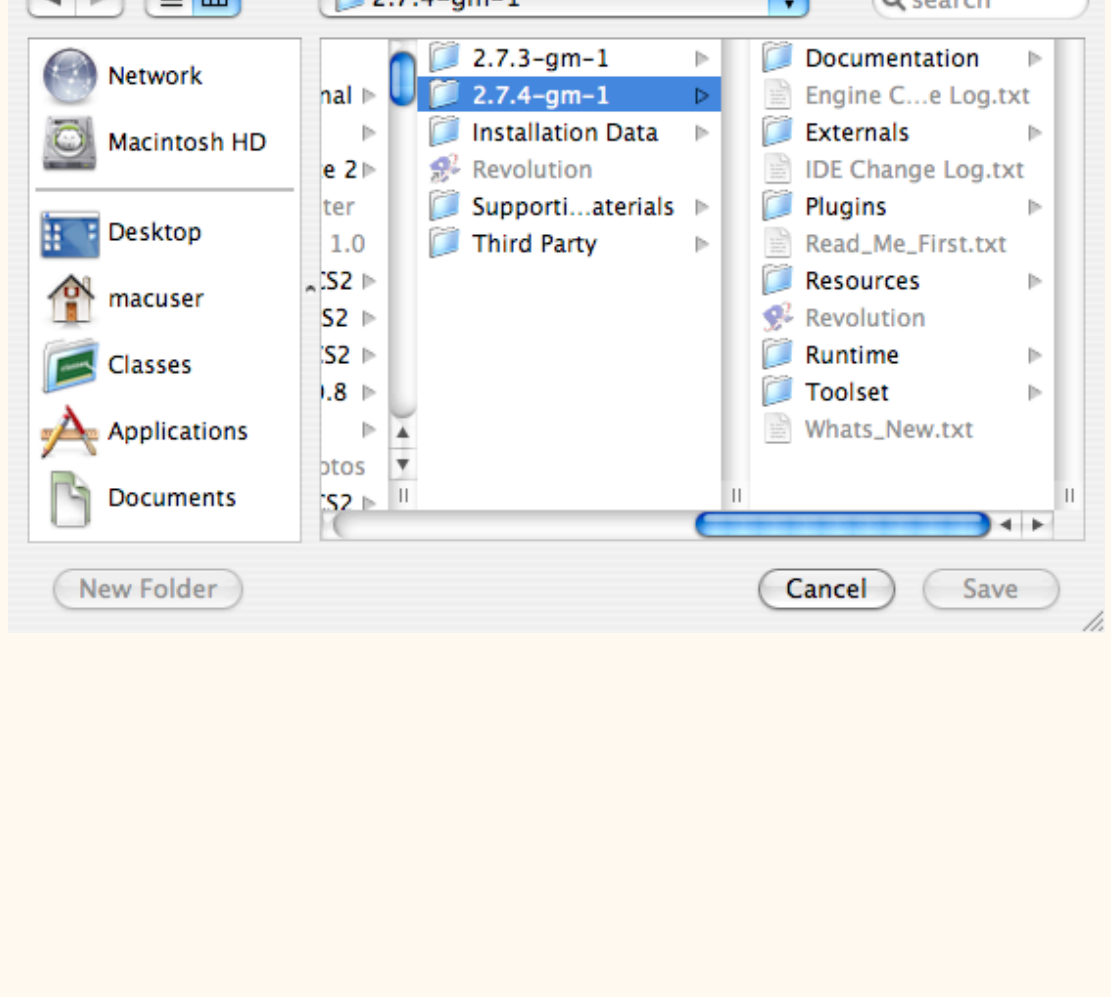
ask file

syntax: `ask file prompt [with defaultFileName]` where *prompt* is a text prompt that will appear in the dialog. If you specify a *defaultFileName*, that string will appear in the entry field for the file name, and the user can choose either to accept it or change it.

The `ask file` command brings up a standard file dialog, such as you see when choosing Save... from the File menu in most applications. It allows the user to type a file name and specify a location to save it,then puts the result into the special variable `it`. Remember that `ask file` does not actually create or write to a file, only returns a pathname and the file name the user typed. The developer can use this information to actually write the data to the file in the location returned by this command.

Here is a sample script segment for using `ask file` for saving some text:

```
ask file "Save file as..."
put it into tFilePath
put fld "myText" into url ("file:" & tSavePath)
```



Examining files and folders in the file system in LiveCode

There are two very useful functions in LiveCode that allow you to examine the contents of a folder in the computer's file system—`the files` and `the folders`.

The `files` function returns a list of files in a folder you designate. By specifying "`detailed`" as the optional second argument, you can also get detailed information about the file, such as its modification date, size, and access permissions.

```
put files(specialFolderPath("desktop")) into fld "myfld"
put files(specialFolderPath("desktop"),"detailed") into fld "myfld"
```

Likewise, the `folders` function returns a list of folders in the designated folder. Again, using the "`detailed`" option gives you detailed information about the folder, such as its creation date and access permissions.

```
put folders(specialFolderPath("resources")) into fld "myfld"
put folders(specialFolderPath("resources"),"detailed") into fld "myfld"
```

The Text Editor Exercise