Learn REBOL

by Nick Antonaccio

Printed in the United States of America
Published by Lulu.com Morrisville, North Carolina USA
Printing History: December 2009 First Edition
ISBN: 978-0-557-22746-4

Contents:

1. Introducing REBOL

What is REBOL? Why use it?

- REBOL is a uniquely small and productive development tool that can be used to create powerful ***desktop* software**, dynamic CGI ***web site*** and server applications, rich distributed *browser plugin* applications, mobile apps, and more. REBOL's blend of capability, compact size, ease of use, cross-platform functionality, and variety of interpreter platforms enable it to gracefully replace many common tools such as Java, Python, Visual Basic, C, C++, PHP, Perl, Ruby, Javascript, toolkits such as wxWidgets, graphic/multimedia platforms such as Flash, DBMSs such as Access, MySQL, and SQLite, a variety of system utilities, and more, all with one simple paradigm. Despite its broad usefulness, REBOL is far **easier** to implement than any other comparable tool.
- REBOL is ***ultra* compact**. Its uncompressed file size is about 1/2 Meg on most platforms. It can be downloaded, installed, and put to use on all supported operating systems in *less than a minute*, even over a slow dialup connection.
- REBOL can also be used immediately, without installation, on over 40 operating systems as a lightweight file manager, text editor, calculator, database manager, email client, ftp client, news reader, image viewer/editor, OS shell, and more. You can use it as a simple utility program with a familiar interface to common computing activities, on just about any computer, even if you're unfamiliar with the operating system.
- REBOL includes *GUI, network, graphics, sound, database, image manipulation, math, parsing, compression, CGI decoding, secure network services, text editing, and other functions built-in*. **No external modules, tool kits, or IDEs are required for any essential functionality**.
- REBOL is ***easy*** enough for absolute beginners and average computer users to operate immediately, but *powerful and rich* enough for a wide variety of complex professional work.
- REBOL has useful *built-in help* for all available functions and language constructs.
- REBOL is supported by a friendly and knowledgeable community of active developers around the world.
- REBOL is available in both ***free*** and supported commercial versions. The free version can be used to create commercial applications, with very few license restrictions. Part of the REBOL language is open source, and that code is available directly in the interpreter. The closed components *are kept in an escrow account*, in case the Rebol Technologies company ever goes out of business (source code escrow licenses are available for those who use it in critical work).
- REBOL has a facility for ultra fast performance using "Rebcode", which can be optimized like assembly language, but works the same way across all supported hardware and operating systems (using the exact same code).
- REBOL was created by Carl Sassenrath, who developed the Amiga operating system executive in 1985 (the first preemptive multitasking OS kernel for personal computers). REBOL has been in commercial use since its first release in 1998, and a 3rd major release of the language is in active development as of 2009.
- REBOL is a modern, multi paradigm development tool (procedural, object oriented, and functional), but its unique syntax goes well beyond traditional approaches to computer language design. REBOL code is typically ***much shorter*** *and more readable* than other languages, and REBOL is often ***far more productive*** than other development tools (very often, *dramatically* so). ***There's absolutely no simpler solution for cross-platform GUI creation, anywhere*** (the code for a complete program window with a button is simply: *view layout [button]*). But that just scratches the surface. REBOL has a striking ability to simplify difficult computing tasks of all types, with straightforward, high level *code dialects*. No other development tool is as adept at creating practical *domain specific languages*. On a more basic level, the storage/manipulation/transfer of all data is managed by a *single ubiquitous code structure*. Arrays, lists, tables, and even sizable databases of mixed text, code, and binary data are all stored using one consistent "block" syntax. Blocks are created simply by surrounding any list of data with square brackets. Like everything else in REBOL, the format is extremely simple, but it enables many powerful features for searching, sorting, comparing, dissecting, evaluating, storing, retrieving, transferring and otherwise manipulating information of all types. Common network protocols and data values are also natively usable in REBOL. You can read and write data directly to/from web servers, email accounts, databases, and more, add/subtract time, date, and other values automatically, manipulate XML, HTML, CSV and other formats natively, display and apply effects to images, play sounds, etc., all without any preparation,

complex formatting, or use of any external library code. REBOL has a built in, powerful "parse" dialect which elegantly replaces the need for regular expressions in most cases. The list of such practical features is long, but REBOL is not built from simple gimmicks - it's a deep and powerful tool. Because so many practical computing elements are all built into REBOL, and interact *natively*, the learning curve required to **get *real work* done** is *much* easier than in other development environments. No other language includes such straightforward and versatile mechanisms for accomplishing the most basic work of computers - managing data of all types.

- REBOL is small, practical, portable, extremely productive, and *different* than the typical mess of modern computing tools. It does *not* rely on a large stack of disparate technologies to accomplish useful computing goals. All it's features exist inside one *tiny* downloadable executable that *anyone* can get running, on just about any computer, in less than a minute. It can be used as anything from compact utility application to powerful professional development environment. Even average computer users with absolutely no coding experience can learn to create *real, useful and powerful* REBOL scripts very quickly.

2. How This Tutorial Is Organized

There are 5 main parts to this text:

1. **Fundamentals:**
   The first sections cover how to use the REBOL interpreter (typing in the console, creating scripts, navigating built-in help, creating .exe's, etc.), basic language constructs and syntax (variables, functions, data types, conditions, loops, i/o, etc.), and the fundamentals of creating GUI program windows.
2. **Examples:**
   9 fully documented programs which demonstrate, line by line, how the above fundamentals are put together to form complete applications.
3. **Other Important Topics:**
   Graphics, animation, 3D, using databases, accessing DLLs and the OS API, web site CGI programming, writing multitasking code, 3rd party tool kits, parse, objects, ports, and more.
4. **Real World Case Studies:**
   21 full case studies covering how a wide variety of complete desktop, network, and web site applications were conceived and created using REBOL. This section demonstrates, step by step, how each of the examples grew from concept to final design using outlines, pseudo code, and detailed finished code. Each example demonstrates a variety of practical REBOL concepts, code patterns, and tools, and helps guide you towards "thinking in REBOL".
5. **Additional Scripts and Resources:**
   More code and resources to help complete your understanding of REBOL and continue learning.

This tutorial covers the REBOL language from the ground up, fully documents the creation of *more than 50 applications*, and contains *many* additional short scripts and useful concepts. Links to other online documentation resources are provided to more fully learn many topics, but no third party reference materials are required to understand any example in this text, even if you've never programmed a line of code before. If you're familiar with other programming languages, be prepared to think about coding in ways that are a bit different from your accustomed patterns. You won't find the *typical* explanations of object oriented programming, modules, pointers, arrays, string management, regular expressions, or other common topics in this text. That's because REBOL's design provides straightforward solutions to reduce or eliminate the need for many complex syntax structures and coding techniques you may know. Every step of the way through this tutorial, you'll pick up practical approaches to easily achieve computing goals of all types. By learning REBOL, you'll learn to get many things *done* more quickly and easily than you can with any other tool. Enjoy!

3. Getting Started: Downloading and Installing REBOL, Hello World

The REBOL interpreter is a program that runs on your computer. It translates written text in the REBOL language syntax ("source code") to instructions the computer understands. To get the free REBOL interpreter, go to:

http://rebol.com/view-platforms.html

For Microsoft Windows, download the rebview.exe file - just click the link with your mouse and save it to your hard drive. If you want to run REBOL on any other operating system (Macintosh, Linux, etc.), just select, download and run the correct file for your computer. It works the same way on every operating system. You can use the stand-alone versions on just about any desktop machine. Upload the correct interpreter version to your web server and you can also execute REBOL CGI programs directly on your web site. You can also install a plugin version to run full REBOL desktop applications directly on pages in a web browser.

Once you've got the tiny REBOL desktop interpreter downloaded, installed, and running on your computer (Start -> Programs -> REBOL -> REBOL View), *click the "Console" icon*, and you're ready to start typing in REBOL programs. To run your first example, type the following line into the REBOL interpreter, and then press the [Enter] (return) key on your keyboard:

```
alert "Hello world!"
```

Before going any further, give it a try. Download REBOL and type in the code above to see how it works. It's extremely simple and literally takes just a few seconds to install. To benefit from this tutorial, type or paste each code example into the REBOL interpreter to see what happens.

(Install Note: on some versions of Linux, you may need to run "./rebview +i" to install the required libs).

4. An Amazingly Tiny Demo and Some Simple Examples

Many of the examples programs in this tutorial are available as downloadable Windows executables, at:

http://musiclessonz.com/rebol_tutorial/examples

To whet your appetite, here's an example that demonstrates just how potent REBOL code can be. The following script contains 10 useful programs in *LESS THAN HALF A PRINTED PAGE OF CODE*:

1. FREEHAND PAINT: Draw and save graphic images
2. SNAKE GAME: Eat the food, avoid hitting the walls and yourself
3. TILE PUZZLE, "15": Arrange the tiles into alphabetical order
4. CALENDAR: Save and view events for any date
5. VIDEO: Live webcam *video* viewer (*not* just a static image)
6. IPs: Display your LAN and WAN IP addresses
7. EMAIL: Read emails from any pop account
8. DAY CALCULATOR: Count the days between 2 selected dates
9. PLAY SOUNDS: Browse your computer for wave files to play
10. FTP TOOL: Web site editor (browse folders on your web server, click files to edit and save changes back to your server, create and edit new files, etc.)

This example is 100% native REBOL code. No external libraries, images, GUI components, or other resources of any kind are imported or called. It runs on Windows, Mac, Linux, and any other OS supported by REBOL/View. To run it, just download the tiny REBOL interpreter and copy/paste the code below into the console (a Windows .exe is also available here):

```
REBOL[title:"Demo"]p: :append kk: :pick r: :random y: :layout q: 'image
z: :if gg: :to-image v: :length? g: :view k: :center-face ts: :to-string
tu: :to-url sh: :show al: :alert rr: :request-date co: :copy g y[style h
btn 150 h"Paint"[g/new k y[s: area black 650x350 feel[engage: func[f a e][
z a = 'over[p pk: s/effect/draw e/offset sh s]z a = 'up[p pk 'line]]]
effect[draw[line]]b: btn"Save"[save/png %a.png gg s al"Saved 'a.png'"]btn
"Clear"[s/effect/draw: co[line]sh s]]]h"Game"[u: :reduce x: does[al join{
SCORE: }[v b]unview]s: gg y/tight[btn red 10x10]o: gg y/tight[btn tan
10x10]d: 0x10 w: 0 r/seed now b: u[q o(((r 19x19)* 10)+ 50x50)q s(((r
19x19)* 10)+ 50x50)]g/new k y/tight[c: area 305x305 effect[draw b]rate 15
feel[engage: func[f a e][z a = 'key[d: select u['up 0x-10 'down 0x10 'left
-10x0 'right 10x0]e/key]z a = 'time[z any[b/6/1 < 0 b/6/2 < 0 b/6/1 > 290
b/6/2 > 290][x]z find(at b 7)b/6[x]z within? b/6 b/3 10x10[p b u[q s(last
b)]w: 1 b/3:((r 29x29)* 10)]n: co/part b 5 p n(b/6 + d)for i 7(v b)1[
either(type?(kk b i)= pair!)[p n kk b(i - 3)][p n kk b i]]z w = 1[clear(
back tail n)p n(last b)w: 0]b: co n sh c]]]do[focus c]]]h"Puzzle"[al{
Arrange tiles alphabetically:}g/new k y[origin 0x0 space 0x0 across style
p button 60x60[z not find[0x60 60x0 0x-60 -60x0]face/offset - x/offset[
exit]tp: face/offset face/offset: x/offset x/offset: tp]p"O"p"N"p"M"p"L"
return p"K"p"J"p"I"p"H"return p"G"p"F"p"E"p"D"return p"C"p"B"p"A"x: p
white edge[size: 0]]]h"Calendar"[do bx:[z not(exists? %s)[write %s ""]rq:
rr g/new k y[h5 ts rq aa: area ts select to-block(find/last(to-block read
%s)rq)rq btn"Save"[write/append %s rejoin[rq" {"aa/text"} "]unview do bx]]
]]h"Video"[wl: tu request-text/title/default"URL:"join"http://tinyurl.com"
"/m54ltm"g/new k y[image load wl 640x480 rate 0 feel[engage: func[f a e][
z a = 'time[f/image: load wl show f]]]]h"IPs"[parse read tu join"http://"
"guitarz.org/ip.cgi"[thru<title>copy my to</title>]i: last parse my none
al ts rejoin["WAN: "i" -- LAN: "read join dns:// read dns://]]h"Email"[
g/new k y[mp: field"pop://user:pass@site.com"btn"Read"[ma: co[]foreach i
read tu mp/text[p ma join i"^/^/^/^/^/^/"editor ma]]]]h"Days"[g/new k y[
btn"Start"[sd: rr]btn"End"[ed: rr db/text: ts(ed - sd)show db]text{Days
Between:}db: field]]h"Sounds"[ps: func[sl][wait 0 rg: load sl wf: 1 sp:
open sound:// insert sp rg wait sp close sp wf: 0]wf: 0 change-dir
```

```
%/c/Windows/media do wl:[wv: co[]foreach i read %.[z %.wav = suffix? i[p
wv i]]]g/new k y[ft: text-list data wv[z wf <> 1[z error? try[ps value][al
"Error"close sp wf: 0]]]btn"Dir"[change-dir request-dir do wl ft/data: wv
sh ft]]]h{FTP}[g/new k y[px: field"ftp://user:pass@site.com/folder/"[
either dir? tu va: value[f/data: sort read tu va sh f][editor tu va]]f:
text-list[editor tu join px/text value]btn"?"[al{Type a URL path to browse
(nonexistent files are created). Click files to edit.}]]]]
```

That's the entire application - all 10 programs. Go ahead, give it a try. Download the REBOL interpreter and copy/paste the code above into the console. It only takes a few seconds. By the end of this tutorial you'll know exactly how all that code works, and much more...

4 - Several Basic Examples

The above example is obfuscated to demonstrate just how malleable and compact REBOL code can be. The following examples represent more typical, readable REBOL code. This first example demonstrates how to create a basic GUI program window (the size info in this example is optional):

```
view layout [size 500x400]
```

Here's a program window with a text area and a button:

```
view layout [area btn "Click Me"]
```

In this example, the button does something when clicked:

```
view layout [
    area
    btn "Click Me" [alert "You can type in the square area."]
]
```

The following example demonstrates how the button can be made to save any text typed into the area, to a file on the hard drive:

```
view layout [
    a: area
    btn "Save" [
        write %reboltut.txt a/text
        alert "Saved"
    ]
]
```

Here's a little text editor application that builds on the idea above. You can likely get a sense of how it works just by glancing through the code:

```
view layout [
    h1 "Text Editor:"
    f: field 600 "filename.txt"
    a: area 600x350
    across
    btn "Load" [
        f/text: request-file
        show f
        a/text: read to-file f/text
```

```
        show a
    ]
    btn "Save" [
        write to-file request-file/save/file f/text a/text
        alert "Saved"
    ]
]
```

Here's an email client you can use to read and send emails to/from any pop/smtp server:

```
view layout[
    h1 "Send:"
    btn "Server settings" [
        system/schemes/default/host: request-text/title "SMTP Server:"
        system/schemes/pop/host:     request-text/title "POP Server:"
        system/schemes/default/user: request-text/title "SMTP User Name:"
        system/schemes/default/pass: request-text/title "SMTP Password:"
        system/user/email: to-email request-text/title "Your Email Addr:"
    ]
    a: field "user@website.com"
    s: field "Subject"
    b: area
    btn "Send"[
        send/subject to-email a/text b/text s/text
        alert "Sent"
    ]
    h1 "Read:"
    f: field "pop://user:pass@site.com"
    btn "Read" [editor read to-url f/text]
]
```

As you can see, REBOL is typically *very* easy to read and write.

4.1 Opening REBOL Directly to the Console

Before typing in or pasting any more code, adjust the following option in the REBOL interpreter: click the "User" menu in the graphic *Viewtop* that opens by default with REBOL, and uncheck "Open Desktop On Startup". That'll save you the trouble of clicking the "Console" button every time you start REBOL.

5. Some Perspective for Absolute Beginners

This tutorial moves at a pace quick enough to satisfy experienced developers, but because REBOL's learning curve is different from other programming languages, it can also be understood clearly by beginners. If you're reading this text as a novice programmer, it can be helpful to understand a few basic concepts that provide perspective about learning to program. First:

*Essentially, all computers do is let users input, store, retrieve, organize, share/transfer, manipulate, alter, view and otherwise deal with **data** in useful ways*.

So, everything you'll do when writing code basically involves manipulating text, numbers, and/or binary data (photos, music, etc.). The fundamental components used to deal with data haven't changed too dramatically in the past few decades. They've simply improved in speed, capacity, and interface. In the current state of modern computing, data is typically input, manipulated, and returned via graphical user interfaces such as program windows, web forms displayed in browsers, and other keyboard/mouse driven "GUI"s. Data is saved on local hard drives and storage devices (CDs, thumb drives, etc.) and on remote web servers, and is typically transferred via local networks and Internet connections. Images, sounds, video, and other types of multimedia data are contained in standardized file formats, and graphic data is displayed using standard mathematical techniques. Knowing how to control those familiar computing elements to allow users to manipulate data, is the goal of learning to program. It doesn't matter whether you're interested in writing business applications to work with inventory and scheduling (text and number data), programs to alter web pages (text and image data), programs to play/edit music (binary data), programs to broadcast video across the Internet (rapidly transferred sequential frames of binary data), programs to control robotic equipment, compute scientific equations, play games, etc... They all require learning to input, manipulate, and return *data* of some sort. You can do all those things with REBOL, and once you've done it in one language, it's easier to do with other programming tools.

REBOL allows programmers to *quickly* build graphic interfaces to input and return all common types of data. It can *easily* manipulate text, graphics, and sounds in useful ways, and it provides *simple* methods to save, retrieve, and share data across all types of hardware, networks, and the Internet. That makes it a great way to begin learning how to program. By learning REBOL, you'll learn about all the fundamental structures and concepts in programming: variables, functions, data types, conditional operations, loops, objects, etc. You'll also learn about important topics such as user interface design, algorithmic thinking, working with databases, the operating system API, CGI, and more. Those topics all share conceptual and technical similarities, regardless of language, and *you'll need to learn to think in those terms* to write computer programs, even in a language that's as easy to learn as REBOL. Despite its ease of use, REBOL is an extremely powerful tool. For years it has been used by professionals in enterprise level work around the world. You may never need to learn another programming language.

If you've never done any real "programming" before, the first part of this text may seem a bit technical. Don't be put off. There is no other language with a faster learning curve than REBOL - you'll begin to see the big picture within a few days. Working through this tutorial, you'll gradually build recognition of REBOL language idioms and practical code patterns, by example. The first part of the tutorial will be a whirlwind introduction to many of the fundamental language elements. Just take it all in, and if you really want to learn, *be sure to type in, or at least copy/paste, each example into the REBOL interpreter*. Reading through the code *isn't* enough.

6. A Quick Summary of the REBOL Language

6.1 Built-In Functions and Basic Syntax

As with any modern programming language, to use REBOL, you need to learn how to use **"functions"**. Functions are words that perform actions. Function words are followed by data **"parameters"** (also called "arguments"). Paste these functions into the REBOL interpreter to see how they work:

```
alert "Alert is a function. THIS TEXT IS ITS PARAMETER."
request "Are you having fun yet?"
editor "Edit this text."
browse http://rebol.com
```

Some functions don't require any data parameters, but do produce **"return"** values. Try these functions in the interpreter. They each return a value selected by the user:

```
request-pass
request-date
request-color
request-file
```

The return values output by the above functions can be used in your programs to accomplish useful goals. The file name output by the "request-file" function, for example, could be used to determine which data gets opened and manipulated in your program. The data returned by the "request-pass" function can be used to control access to selected data.

Many functions have **optional or limited parameters/return values**. These options, called "refinements", are specified by the "/" symbol. Try these variations of the "request-pass" function to see how they each perform differently:

```
request-pass/only
request-pass/user "username"
request-pass/title "The 'title' refinement sets this header text."
request-pass/offset/title 10x100 "'offset' repositions the requester."
```

Some functions take **multiple arguments**. The "rejoin" function returns the joined ("concatenated") text arguments inside brackets. *Concatenation is very important in all types of programming* - you will see this function in use often:

```
rejoin ["Hello " "there" "!"]
```

6.1.1 Understanding Return Values and the Order of Evaluation

In REBOL, you can put as many functions as you want on one line, and *they are all evaluated strictly from left to right*. Functions are grouped together automatically with their required data parameter(s). The following line contains two alert functions:

```
alert "First function" alert "Second function"
```

Rebol knows to look for one parameter after the first alert function, so it uses the next piece of data on that line as the argument for that function. Next on the line, the interpreter comes across another alert function, and uses the following text as it's data parameter.

In the following line, the first function "request-pass/offset/title" requires *two parameters*, so REBOL uses *the next two items on the line* ("10x100" and "title") as its arguments. After that's complete, the interpreter comes across another "alert" function, and uses the following text, "Processing", as its argument:

```
request-pass/offset/title 10x100 "title" alert "Processing"
```

IMPORTANT: In REBOL, the return values (output) from one function can be used directly as the arguments (input) for other functions. Everything is simply evaluated from left to right. In the line below, the "alert" function takes *the next thing on the line as it's input parameter*, which in this case is *not a piece of data*, but a *function which returns some data* (the concatenated text returned by the "rejoin" function):

```
alert rejoin ["Hello " "there" "!"]
```

To say it another way, the value returned above by the "rejoin" function is passed to (used as a parameter by) the "alert" function. Parentheses can be used to clarify which expressions are evaluated and passed as parameters to other functions. The parenthesized line below is treated by the REBOL interpreter exactly the same as the line above - it just lets you see more clearly what data the "alert" function puts on screen:

```
alert ( rejoin ["Hello " "there" "!"] )
```

Perhaps the hardest part of getting started with REBOL is understanding the order in which functions are evaluated. The process can appear to work backwords at times. In the example below, the "editor" function takes the next thing on the line as it's input parameter, and edits that text. In order for the editor function to begin its editing operation, however, it needs a text value to be returned from the "request-text" function. The first thing the user *sees* when this line runs, therefore, is the text requester. That appears backwards, compared to the way it's written:

```
editor (request-text)
```

Always remember that lines of REBOL code are evaluated from left to right. If you use the return value of one function as the argument for another function, the execution of the whole line will be held up until the necessary return value is processed.

Any number of functions can be written on a single line, with return values cascaded from one function to the next:

```
alert ( rejoin ( ["You chose: " ( request "Choose one:" ) ] ) )
```

The line above is typical of common REBOL language syntax. There are three functions: "alert", "rejoin", and "request". In order for the first alert function to complete, it needs a return value from "rejoin", which in turn needs a return value from the "request" function. The first thing the user sees, therefore, is the request function. After the user responds to the request, the selected response is rejoined with the text "You chose: ", and the joined text is displayed as an alert message. Think of it as reading "display (the following text joined together ("you chose" (an answer selected by the user))). To complete the line, the user must first answer the question.

To learn REBOL, it's essential to first memorize and recognize REBOL's many built-in function words, along with the parameters they accept as input, and the values which they return as output. When you get used to reading lines of code as functions, arguments, and return values, read from left to right, the language will quickly begin to make sense.

It should be noted that in REBOL, math expressions are evaluated from *left to right* like all other functions. There is no "order of precedence", as in other languages (i.e., multiplication doesn't automatically get computed before addition). To force a specific order of evaluation, enclose the functions in parentheses:

```
print  (10  +  12) /  2      ;  22 / 2 = 11  (same as without parentheses)
print   10  + (12  /  2)     ;  10 + 6 = 16
```

REBOL's left to right evaluation is simple and consistent. Parentheses can be used to clarify the flow of code, if ever there's confusion.

6.1.2 White Space

Unlike other languages, REBOL *does **not** require any line terminators* between expressions (functions, parameters, etc.), and you can insert empty white space (tabs, spaces, newlines, etc.) as desired into code. Text after a semicolon and before a new line is treated as a comment (ignored entirely by the interpreter). The code below works exactly the same as the previous example. *Notice that tabs are used to indent the block of code inside the square brackets* and that the contents of the brackets are spread across multiple lines. This helps group the code together visually, but it's not required:

```
alert rejoin [
    "You chose: "               ; 1st piece of joined data
    (request "Choose one:")     ; 2nd piece of joined data
]
```

ONE CAVEAT: parameters for most functions should begin on the same line as the function word. The following example will *not* work properly because the rejoin arguments' opening brackets need to be on the same line as the rejoin function:

```
alert rejoin                    ; This does NOT work.
[                               ; Put this bracket on the line above.
    "You chose: "
    (request "Choose one:")
]
```

If you want to comment out a large section of code, simply surround it with curly braces:

```
{
    This line doesn't do anything.
    This line also does nothing.
    This line is ignored too.
}
```

6.2 More Basics: Word Assignment, I/O, Files, Built-In Data Types and Native Protocols

In REBOL, *the colon (":") symbol is used to assign word labels ("variables") to values*:

```
person: "John"
```

Now, the word label "person" can be used anywhere (without the colon), to represent the text "John". Notice that the variable "person" has been rejoined with some other text below:

```
alert rejoin ["The person's name is " person]
```

Word labels are NOT case sensitive:

```
alert person
alert PERSON
alert PeRsOn

; to the REBOL interpreter, all three lines above are the same
```

In this next example, the word "filename" is assigned to *the value returned by the request-file function* (a file chosen by the user):

```
filename: request-file
```

Now, the label "filename" can be used to represent the *file selected above*:

```
alert rejoin ["You chose " filename]
```

REBOL is a bit different from other programming languages in that word labels can be assigned to *anything*: numbers, text strings, binary data, arrays, lists, hash tables, functions, and even executable blocks of code. At this point, just be aware that when you see the colon symbol, a word label is being assigned to some value.

The "ask" function is a simple way to get some text data from a user at the interpreter's command line (similar to "request-text", but without using a pop-up requester):

```
ask "What is your name? "
```

In the example below, the variable "name" is assigned to *the text **returned** by the ask function* (i.e., entered by the user). Again, the parentheses are not required - they're just there to clarify the grouping together of the 'ask' function with its text argument:

```
name: (ask "What is your name? ")
```

Now you can use the variable word "name" to represent whatever text the user typed in response to the above question.

The "print" function is a simple way to display text data at the interpreter's command line:

```
print rejoin ["Good to meet you " name]
```

The "prin" function prints consecutive text elements right next to each other (not on consecutive lines):

```
prin "All " prin "on " prin "one " print "line." print "On another."
```

Multi-line formatted text is enclosed in curly braces ("{}"), instead of quotes:

```
print {
    Line 1
    Line 2
    Line 3
}
```

Quotes and curly braces can be used interchangeably on a single line:

```
print {"text"}
print "{text}"
```

You can print a carriage return using the word "newline" or the characters ^/

```
print rejoin ["This text if followed by a carriage return." newline]
print "This text if followed by a carriage return.^/"
```

Clear the screen using "newpage":

```
prin newpage
```

The "write" function saves data to a file. It takes **two parameters**: a file name to write to, and some data to write to that file.

```
write %/C/YOURNAME.txt name
```

NOTE: in REBOL, the percent character ("%") is used to represent local files. Because REBOL can be used on many operating systems, and because those operating systems all use different syntax to refer to drives, paths, etc., REBOL uses the universal format: %/drive/path/path/.../file.ext . For example, "%/c/windows/notepad.exe" refers to "C:\Windows\Notepad.exe" in Windows. REBOL converts that syntax to the appropriate operating system format, so that your code can be written once and used on every operating system, without alteration. The following 2 functions convert REBOL file format to your operating system's format, and visa versa:

```
to-local-file %/C/YOURNAME.txt
to-rebol-file "C:\YOURNAME.txt"
```

You can *write data to a web site* (or any other connected protocol) using the exact same write syntax that is used to write to a file (be sure to use an appropriate username and password for your web site ftp account):

```
write ftp://user:pass@website.com/name.txt name
```

The "read" function reads data from a file:

```
print (read %/C/YOURNAME.txt)
```

REBOL has a built-in text editor that can also read, write, and manipulate text data:

```
editor %/c/YOURNAME.txt
```

You can read data straight from a web server, an ftp account, an email account, etc. using the same format. Many Internet protocols are built right into the REBOL interpreter. They're understood natively, and REBOL knows exactly how to connect to them without any preparation by the programmer:

```
editor http://rebol.com                         ; Reads the content of the
                                                ;   document at this URL.
editor pop://user:pass@website.com              ; Reads all emails in this
                                                ;   POP inbox.
editor clipboard://                             ; Reads data that has
                                                ;   been copied/pasted to
                                                ;   the OS clipboard.
print read dns://msn.com                        ; Displays the DNS info
                                                ;   for this address.
print read nntp://public.teranews.com           ; (Hit the [ESC] key to stop
                                                ;   this Usenet listing.)

; NOTE: The editor reads, AND allows you to SAVE EDITS back to the server:

editor ftp://user:pass@website.com/public_html/index.html
```

Transferring data between devices connected by any supported protocol is easy - *just read and write*:

```
; read data from a web site, and paste it into the local clipboard:

write clipboard:// (read http://rebol.com)    ; afterward, try pasting into
                                              ; your favorite text editor

; read a page from one web site, and write it to another:

write ftp://user:pass@website2.com (read http://website1.com)

; again, notice that the "write" function takes TWO parameters
```

Sending email is just as easy, using a similar syntax:

```
send user@website.com "Hello"
send user@website.com (read %file.txt)        ; sends an email, with
                                              ; file.txt as the body
```

The "**/binary**" modifier is used to read or write binary (non-text) data. *You'll use read/binary and write/binary to read and write images, sounds, videos and other non-text files*:

```
write/binary %/c/bay.jpg read/binary http://rebol.com/view/bay.jpg
```

For clarification, remember that the write function takes two parameters. The first parameter above is "%/c/bay.jpg". The second parameter is the binary data read from http://rebol.com/view/bay.jpg:

```
write/binary (%/c/bay.jpg) (read/binary http://rebol.com/view/bay.jpg)
```

The "load" and "save" functions also read and write data, but in the process, *automatically format* certain data types for use in REBOL. Try this:

```
; assign the word "picture" to the image "load"ed from a given URL:

picture: load http://rebol.com/view/bay.jpg

; save the image to a given file name, and automatically convert it
; to .png format;

save/png %/c/picture.png picture

; show it in a GUI window (much more about this in the next section):

view layout [image load %/c/picture.png]
```

"Load" and "save" are used to conveniently manage certain types of data in formats directly usable by REBOL (images, sounds, DLLs, certain native data structures, etc. can be loaded and used immediately). You'll use "read" and "write" more commonly to store and retrieve typical types of data, exactly byte for byte, to/from a storage medium, when no conversion or formatting is necessary.

REBOL automatically knows how to perform appropriate computations on times, dates, IP addresses, coordinate values, and other common types of data:

```
print 3:30am + 00:07:19                   ; increment time values properly
print now                                 ; print current date and time
print now + 0:0:30                        ; print 30 seconds from now
print now - 10                            ; print 10 days ago
print 23x54 + 19x31                       ; easily add coordinate pairs
print 192.168.1.1 + 000.000.000.37        ; easily increment ip addresses
view layout [image picture effect [flip]] ; apply effects to image types
```

REBOL also natively understands how to use URLs, email addresses, files/directories, money values, tuples, hash tables, sounds, and other common values in expected ways, *simply by the way the data is formatted*. You don't need to declare, define, or otherwise prepare such types of data as in other languages - just use them.

```
To determine the type of any value, use the "type?" function:

some-text:  "This is a string of text"    ; strings of text go between
type? some-text                           ; "quotes" or {curly braces}

an-integer: 3874904                       ; integer values are just pos-
type? an-integer                          ; itive/negative whole numbers

a-decimal: 7348.39                        ; decimal numbers are recognized
type? a-decimal                           ; by the decimal point
```

```
web-site: http://musiclessonz.com          ; URLs are recognized by the
type? web-site                             ; http://, ftp://, etc.

email-address: user@website.com            ; email values are in the
type? email-address                        ; format user@somewebsite.domain

the-file: %/c/myfile.txt                   ; files are preceded by the %
type? the-file                             ; character

bill-amount: $343.56                       ; money is preceded by the $
type? bill-amount                          ; symbol

html-tag: <br>                             ; tags are places between <>
type? html-tag                             ; characters

binary-info:  #{ddeedd}                    ; binary data is put between
type? binary-info                          ; curly braces and preceded by
                                           ; the pound symbol

image: load http://rebol.com/view/bay.jpg ; REBOL can even automatically
type? image                                ; recognize the data type of
                                           ; most common image formats.

a-sound: load %/c/windows/media/tada.wav   ; And sounds too!
a-sound/type
```

Data types can be specifically "cast" (created, or assigned to different types) using "to-(type)" functions:

```
numbr: 4729                ; The label 'numbr now represents the integer
                           ;   4729.
strng: to-string numbr     ; The label 'strng now represents a piece of
                           ;   quoted text made up of the characters
                           ;   "4729".  Try adding strng + numbr, and
                           ;   you'll get an error.

; This example creates and adds two coordinate pairs.  The pairs are
; created from individual integer values, using the "to-pair" function:

x: 12  y: 33  q: 18  p: 7
pair1: to-pair rejoin [x "x" y]        ; 12x33
pair2: to-pair rejoin [q "x" p]        ; 18x7
print pair1 + pair2                    ; 12x33 + 18x7 = 30x40

; This example builds and manipulates a time value using the "to-time"
; function:

hour: 3
minute: 45
second: 00
the-time: to-time rejoin [hour ":" minute ":" second]    ; 3:45am
later-time: the-time + 3:00:15
print rejoin ["3 hours and 15 seconds after 3:45 is " later-time]

; This converts REBOL color values (tuples) to HTML colors and visa versa:

to-binary request-color
to-tuple #{00CD00}
```

REBOL has many built-in helper functions for dealing with common data types. Another way to create pair values is with the "as-pair" function. You'll see this sort of pair creation commonly in

games which plot graphics at coordinate points on the screen:

```
x: 12  y: 33  q: 18  p: 7
print (as-pair x y) + (as-pair q p)    ; much simpler!
```

Built-in network protocols, native data types, and consistent language syntax for reading, writing, and manipulating data allow you to perform common coding chores easily and intuitively in REBOL. Remember to type or paste every example into the REBOL interpreter to see how each function and language construct operates.

6.3 GUIs (Program Windows)

Graphic user interfaces ("GUI"s) are easier to create in REBOL than in any other language. The functions "view" and "layout" are used together to display GUIs. *The parameters passed to the layout function are enclosed in brackets*. Those brackets can include identifiers for all types of GUI elements ("widgets"):

```
view layout [btn]  ; creates a GUI with a button

view layout [field]  ; creates a GUI with a text input field

view layout [text "REBOL is really pretty easy to program"]

view layout [text-list]  ; a selection list

view layout [
    button
    field
    text "REBOL is really pretty easy to program."
    text-list
    check
]
```

In REBOL, widgets are called "styles", and the entire GUI dialect is called "VID". You can adjust the visual characteristics of any style in VID by following it with appropriate modifiers:

```
view layout [
    button red "Click Me"
    field "Enter some text here"
    text font-size 16 "REBOL is really pretty easy to program." purple
    text-list 400x300 "line 1" "line 2" "another line"
    check yellow
]
```

The size of your program window can be specified by either of these two formats:

```
view layout [size 400x300]
view layout/size [] 400x300

; both these lines do exactly the same thing
```

A variety of functions are available to control the alignment, spacing, and size of elements in a GUI layout:

```
view layout [
    size 500x350
    across
    btn "side" btn "by" btn "side"
    return
    btn "on the next line"
    tab
    btn "over a bit"
    tab
    btn "over more"
    below
    btn 160 "underneath" btn 160 "one" btn 160 "another"
```

```
    at 359x256
    btn "at 359x256"
]
```

VERY IMPORTANT: You can have widgets perform functions when clicked, or when otherwise activated. Just put the functions inside another set of brackets *after* the widget. This is how you get your GUIs to *'do something'* (using the fundamentals introduced in the previous section):

```
view layout [button "click me" [alert "You clicked the button."]]

view layout [btn "Display Rebol.com HTML" [editor read http://rebol.com]]

view layout [btn "Write current time to HD" [write %time.txt now/time]]

; The word "value" refers to data contained in a currently activated
; widget:

view layout [
    text "Some action examples.  Try using each widget:"
    button red "Click Me" [alert "You clicked the red button."]
    field 400 "Type some text here, then press [Enter] on your keyboard" [
        alert value
    ]
    text-list 400x300 "Select this line" "Then this line" "Now this one" [
        alert value
    ]
    check yellow [alert "You clicked the yellow check box."]
    button "Quit" [quit]
]
```

To react to right-button mouse clicks on a widget, put the functions to be performed inside a *second* set of brackets after the widget:

```
view layout [
    btn "Right Click Me" [alert "left click"][alert "right click"]
]
```

You can assign keyboard shortcuts (keystrokes) to any widget, so that pressing the key reacts the same way as activating the GUI widget:

```
view layout [
    btn "Click me or press the 'A' key on your keyboard" #"a" [
        alert "You just clicked the button OR pressed the 'A' key"
    ]
]
```

You can assign a *word label* to any widget, and refer to data and properties of that widget by its label. The "text" property is especially useful:

```
view layout [
    page-to-read: field "http://rebol.com"
    ; page-to-read/text now refers to the text contained in that field
    btn "Display HTML" [editor read (to-url page-to-read/text)]
]
```

You can also *set* various properties of a widget using its assigned label. When the "Edit HTML Page"

button is clicked below, the text of the multi-line area widget is set to contain the text read from the given URL. The "show" function in the example below is *very* important. It must be used to update the GUI display any time a widget property is changed (if you ever create a GUI that doesn't seem to respond properly, the first thing to check is that you've used a "show" function to properly update any changes on screen):

```
view layout [
    page-to-read: field "http://rebol.com"
    the-html: area 600x440
    btn "Download HTML Page" [
        the-html/text: read (to-url page-to-read/text)
        ; try commenting out the following line to see what happens:
        show the-html
    ]
]
```

Below are two more examples of the above code pattern (getting and setting a widget's text property) - it's a very important idiom in REBOL GUIs. In the first example, the variable "f" is assigned to the field widget, then the variable "t" is assigned to the text contained in that field. In the second example, "t" is assigned the text contained in the f1 field, then the text in f2 is set to "t" - again, *all using the colon symbol*. Note the use of the "show" function to update the display:

```
view layout [
    f: field
    btn "Display Variable" [

    ; When the button is pressed, set the variable
    ; "t" to hold the text currently in the field
    ; above, then alert the contents of that variable:

        t: f/text
        alert t
    ]
]

view layout [
    f1: field
    btn "Display Variable" [

        ; Set the variable "t" to the text contained
        ; in the f1 field above:

        t: f1/text

        ; Now CHANGE the text in the f2 below to
        ; to equal the text stored in variable "t":

        f2/text: t
        show f2
    ]
    f2: field
]

; You GET the text from a widget by assigning a VALUE to equal the
; widget's text property.  You SET/CHANGE the text of a widget by
; assigning THE TEXT PROPERTY of that widget to equal a value.
```

The "offset" of a widget holds its coordinate position. It's another useful property, especially for GUIs which involve movement:

```
view layout [
    size 600x440
    jumper: button "click me" [
        jumper/offset: random 580x420
        ; The "random" function creates a random value within
        ; a specified range.  In this example, it creates a
        ; random coordinate pair within the range 580x420,
        ; every time the button is clicked.  That random value
        ; is assigned to the position of the button.
    ]
]
```

The "style" function is very powerful. It allows you to assign a specific widget definition, *including all its properties and actions*, to any word label you choose. Any instance of that word label is thereafter treated as a replication of the entire widget definition:

```
view layout [
    size 600x440
    style my-btn btn green "click me" [
        face/offset: random 580x420
    ]
    ; "my-btn" now refers to all the above code
    at 254x84 my-btn
    at 19x273 my-btn
    at 85x348 my-btn
    at 498x12 my-btn
    at 341x385 my-btn
]
```

REBOL is great at dealing with all types of common data - not just text. You can easily display photos and other graphics in your GUIs, play sounds, display web pages, etc. Here's some code that downloads an image from a web server and displays it in a GUI - notice the "view layout" functions again:

```
view layout [image (load http://rebol.com/view/bay.jpg)]
```

The "image" widget inside the brackets displays a picture (.bmp, .jpg, .gif., .png) in the GUI. The "load" function downloads the image to be displayed.

REBOL can apply many built-in effects to images:

```
view layout [image (load http://rebol.com/view/bay.jpg) effect [Emboss]]

view layout [image (load http://rebol.com/view/bay.jpg) effect [Flip 1x1]]

; The parentheses are not required:

view layout [image load http://rebol.com/view/bay.jpg effect [Grayscale]]

; There are MANY more built-in effects.
```

You can impose images onto most types of widgets:

```
view layout [area load http://rebol.com/view/bay.jpg]
```

```
; Use the "fit" effect to stretch or shrink the size of the image to that
; of the widget:

view layout [area load http://rebol.com/view/bay.jpg effect [Fit]]
view layout [button load http://rebol.com/view/bay.jpg effect [Fit]]
view layout [field load http://rebol.com/view/bay.jpg effect [Fit Emboss]]

; You can still type into the field and area widgets, as usual.
```

You can apply colors directly to images, just like any other widget. Notice that you can perform calculations directly on color values:

```
view layout [image load http://rebol.com/view/bay.jpg yellow]
view layout [image load http://rebol.com/view/bay.jpg (yellow / 2)]
view layout [image load http://rebol.com/view/bay.jpg (yellow + 0.0.132)]
```

Color gradients (fades from one color to another) are also simple to apply to any widget:

```
view layout [area effect [gradient red blue]]
view layout [
    size 500x400
    backdrop effect [gradient 1x1 tan brown]
    box effect [gradient 123.23.56 254.0.12]
    box effect [gradient blue gold/2]
]
```

You can assign a word label to any layout of GUI widgets, and then display those widgets simply by using the assigned word:

```
gui-layout1: [button field text-list]
view layout gui-layout1
```

You can save any GUI layout as an image, using the "to-image" function. *This enables a built in screen shot mechanism*, and also allows you to easily create/save/manipulate new images using any of the graphic capabilities in REBOL:

```
; assign the label "picture" to an image of a layout:

picture: to-image layout [
    page-to-read: field "http://rebol.com"
    btn "Display HTML"
]

; save it to the hard drive as a .png file:

save/png %/c/layout.png picture
```

Here are some other GUI elements used in REBOL's "VID" layout language:

```
view layout [
    backcolor white
    h1 "More GUI Examples:"
    box red 500x2
    bar: progress
```

```
    slider 200x16 [bar/data: value show bar]
    area "Type here"
    drop-down
    across
    toggle "Click" "Here" [print value]
    rotary "Click" "Again" "And Again" [print value]
    choice "Choose" "Item 1" "Item 2" "Item 3" [print value]
    radio radio radio
    led
    arrow
    return
    text "Normal"
    text "Bold" bold
    text "Italic" italic
    text "Underline" underline
    text "Bold italic underline" bold italic underline
    text "Serif style text" font-name font-serif
    text "Spaced text" font [space: 5x0]
    return
    h1 "Heading 1"
    h2 "Heading 2"
    h3 "Heading 3"
    h4 "Heading 4"
    tt "Typewriter text"
    code "Code text"
    below
    text "Big" font-size 32
    title "Centered title" 200
    across
    vtext "Normal"
    vtext "Bold" bold
    vtext "Italic" italic
    vtext "Underline" underline
    vtext "Bold italic underline" bold italic underline
    vtext "Serif style text" font-name font-serif
    vtext "Spaced text" font [space: 5x0]
    return
    vh1 "Video Heading 1"
    vh2 "Video Heading 2"
    vh3 "Video Heading 3"
    vh4 "Video Heading 3"
    label "Label"
    below
    vtext "Big" font-size 32
    banner "Banner" 200
]
```

Here's a list of all the built in widgets (remember, in REBOL's VID language, widgets are called "styles"):

```
probe extract svv/vid-styles 2
```

Here's a list of the changeable attributes ("facets") available to all widgets:

```
probe remove-each i copy svv/facet-words [function? :i]
```

And here's a list of available layout words:

```
probe svv/vid-words
```

That's just the tip of the iceberg. With REBOL, even absolute beginners can create nice looking, powerful graphic interfaces in minutes. See http://rebol.com/docs/easy-vid.html and http://rebol.com/docs/view-guide.html for more information. Here's a little game that demonstrates common GUI techniques (this whole program was presented earlier in the demo app, in a more compact format without any comments. It's tiny.):

```
; Create a GUI that's centered on the user's screen:

view center-face layout [

    ; Define some basic layout parameters.  "origin 0x0"
    ; starts the layout in the upper left corner of the
    ; GUI window.  "space 0x0" dictates that there's no
    ; space between adjacent widgets, and "across" lays
    ; out consecutive widgets next to each other:

    origin 0x0 space 0x0 across

    ; The section below creates a newly defined button
    ; style called "piece", with an action block that
    ; swaps the current button's position with that of
    ; the adjacent empty space.  That action is run
    ; whenever one of the buttons is clicked:

    style piece button 60x60 [

        ; The line below checks to see if the clicked button
        ; is adjacent to the empty space.  The "offset"
        ; refinement contains the position of the given
        ; widget.  The word "face" is used to refer to the
        ; currently clicked widget.  The "empty" button is
        ; defined later (at the end of the GUI layout).
        ; It's ok that the empty button is not yet defined,
        ; because this code is not evaluated until the
        ; the entire layout is built and "view"ed:

        if not find [0x60 60x0 0x-60 -60x0
            ] (face/offset - empty/offset) [exit]

        ; In English, that reads 'subtract the position of
        ; the empty space from the position of the clicked
        ; button (the positions are in the form of
        ; Horizontal x Vertical coordinate pairs).  If that
        ; difference isn't 60 pixels on one of the 4 sides,
        ; then don't do anything.' (60 pixels is the size of
        ; the "piece" button defined above.)

        ; The next three lines swap the positions of the
        ; clicked button with the empty button.

        ; First, create a variable to hold the current
        ; position of the clicked button:

        temp: face/offset

        ; Next, move the button's position to that of the
        ; current empty space:
```

```
        face/offset: empty/offset

        ; Last, move the empty space (button), to the old
        ; position occupied by the clicked button:

        empty/offset: temp
    ]

    ; The lines below draw the "piece" style buttons onto
    ; the GUI display.  Each of these buttons contains all
    ; of the action code defined for the piece style above:

    piece "1"   piece "2"   piece "3"   piece "4" return
    piece "5"   piece "6"   piece "7"   piece "8" return
    piece "9"   piece "10"  piece "11"  piece "12" return
    piece "13"  piece "14"  piece "15"

    ; Here's the empty space.  Its beveled edge is removed
    ; to make it look less like a movable piece, and more
    ; like an empty space:

    empty: piece 200.200.200 edge [size: 0]
]
```

Advanced users may be interested in understanding why the two words "view" and "layout" are used to create GUIs. Those functions represent two complete and separate language dialects in REBOL. The "view" function is a front end to the lower level graphic compositing engine and user interface system built into REBOL. "Layout" is a higher level function that simply assembles view functions required to draw and manipulate common GUI elements. Understanding how the two operate under the hood is helpful in understanding just how deep, compact, and powerful the REBOL language and dialecting design is. For more information, see http://rebol.com/docs/view-system.html.

6.4 Blocks, Series, and Strings

In REBOL, all *multiple pieces of grouped data items* are stored in "blocks". Blocks are delineated by starting and ending brackets:

```
[ ]
```

Data items in blocks are *separated by white space*. Here's a block of text items:

```
["John" "Bill" "Tom" "Mike"]
```

Blocks were snuck in earlier as multiple text arguments passed to the "rejoin" function, and as brackets used to delineate GUI code passed to the 'view layout' functions:

```
rejoin ["Hello " "there!"]
view layout [button "Click Me" [alert "Hello there!"]]
```

Blocks are actually the fundamental structure used to organize REBOL *code*. You'll find brackets throughout the language syntax to delineate functions, parameters, and other items. In the next section of this tutorial, you'll see more about functions and control structures that use brackets to separate grouped items of code. This section will cover how *data* can be grouped into blocks.

The key concept to understand with blocks is that they are used to hold *multiple* pieces of data. Like any other variable data, blocks can be assigned word labels:

```
some-names: ["John" "Bill" "Tom" "Mike"]

; "some-names" now refers to all 4 of those text items

print some-names
```

Blocks of text data (lists) can be displayed in GUIs, using the "text-list" widget:

```
view layout [text-list data (some-names)]
```

The "append" function is used to add items to a block:

```
append some-names "Lee"
print some-names

append gui-layout1 [text "This text was appended to the GUI block."]
view layout gui-layout1
```

The "foreach" function is used to do something to/with each item in a block:

```
foreach item some-names [alert item]
```

The "remove-each" function can be used to remove items from a block that match a certain criteria:

```
remove-each name some-names [find name "i"]
```

```
; removes all names containing the letter "i" - returns ["John" "Tom"]
```

Empty data blocks are created with the "copy" function. *"Copy" assures that blocks are erased and defined without any previous content*. You'll use "copy" whenever you need to create an empty block:

```
; Create a new empty block like this:

empty-block: copy []

; NOT like this:

empty-block: []
```

Here's a very typical example that uses a block to save text entered into the fields of a GUI. When the "Save" button is pressed, the text in each of the fields is appended to a new empty block, then that whole block is saved to a text file. To later retrieve the saved values, the block is loaded from the text file, and its items assigned back to the appropriate fields in the GUI:

```
view gui: layout [

    ; label some text fields:

    field1: field
    field2: field
    field3: field

    ; add a button:

    btn "Save" [

        ; when the button is clicked, create a new empty block:

        save-block: copy []

        ; add the text contained in each field to the block:

        append save-block field1/text
        append save-block field2/text
        append save-block field3/text

        ; save the block to a file:

        save %save.txt save-block
        alert {SAVED -- Now try running this script again, and load
            the data back into the fields.}
    ]

    ; another button:

    btn "Load" [

        ; load the saved block:

        save-block: load %save.txt

        ; set the text in each field to the contents of the block:

        field1/text: save-block/1
```

```
        field2/text: save-block/2
        field3/text: save-block/3

        ; update the GUI display:

        show gui
    ]
]
```

After running the script above, open the save.txt file with a text editor, and you'll see it contains the text from the fields in the GUI. You can edit the save.txt file with your text editor, then click the "Load" button, and the edited values will appear back in the GUI. You'll use blocks regularly to store and retrieve *multiple pieces* of data in this way, using text files.

6.4.1 Series Functions

In REBOL, blocks can be automatically treated as lists of data, called "series", and manipulated using built-in functions that enable searching, sorting, and otherwise organizing the blocked data:

```
some-names: ["John" "Bill" "Tom" "Mike"]

sortednames: sort some-names        ; sort alphabetically/ordinally

print first sortednames             ; displays the first item ("Bill")

print sortednames/1                 ; ALSO displays the first item ("Bill")
                                    ; (just an alternate syntax)

print pick sortednames 1            ; ALSO displays the first item ("Bill")
                                    ; (another alternate syntax)

find some-names "John"              ; SEARCH for "John" in the block,
                                    ;  set a position marker after that
                                    ;  item - a very important function

find/last some-names "John"         ; search for "John" backwards from
                                    ;  the end of the block

select some-names "John"            ; search for "John" in the block
                                    ;  and return the Next item.

reverse sortednames                 ; reverse the order of items in the
                                    ;  block

length? sortednames                 ; COUNT items in the block - important

head sortednames                    ; set a position marker at the
                                    ;  beginning of the block

next sortednames                    ; set a position marker at the next
                                    ;  item in the block

back sortednames                    ; set a position marker at the
                                    ;  previous item in the block

last sortednames                    ; set a position marker at the last
                                    ;  item in the block

tail sortednames                    ; set a position marker after the
                                    ;  last item in the block
```

```
at sortednames x                    ; set a position marker at the x
                                    ;  numbered item in the block

skip sortednames x                  ; set a position marker x items
                                    ;  forward or backward in the block

extract sortednames 3               ; collect every third item from the
                                    ;  block

index? sortednames                  ; retrieves position number of the
                                    ;  currently marked item in the block

insert sortednames "Lee"            ; add the name "Lee" at the current
                                    ;  position in the block

append sortednames "George"         ; add "George" to the tail of the block
                                    ;  and set position marker to the head

remove sortednames                  ; remove the item at the currently
                                    ;  marked position in the block

remove find sortednames "Mike"      ; ... find the "Mike" item in the
                                    ;  block and remove it

change sortednames "Phil"           ; change the item at the currently
                                    ;  marked position to "Phil"

change third sortednames "Phil"     ; change the third item to "Phil"

poke sortednames 3 "Phil"           ; another way to change the third item
                                    ;  to "Phil"

copy/part sortednames 2             ; get the first 2 items in the block

clear sortednames                   ; remove all items in the block after
                                    ;  the currently marked position

replace/all sortednames "Lee" "Al" ; replace all occurrences of "Lee" in
                                    ;  the block with "Al"

both: join some-names sortednames   ; concatenate both blocks together

intersect sortednames some-names    ; returns the items found in both
                                    ;  blocks

difference sortednames some-names   ; returns the items that are NOT
                                    ;  found in BOTH blocks

exclude sortednames some-names      ; returns the items in sortednames that
                                    ;  are NOT also in some-names

union sortednames some-names        ; returns the items found in both
                                    ;  blocks, ignoring duplicates

unique sortednames                  ; returns all items in the block,
                                    ;  with duplicates removed

empty? sortednames                  ; returns true if the block is empty

write %/c/names.txt some-names      ; write the block to the hard drive
                                    ;  as raw text data

save %/c/namess.txt some-names      ; write the block to the hard drive
```

```
                                        ;  as native REBOL formatted code
```

Learning to use series functions is *absolutely fundamental* to using REBOL. They will be covered by example throughout this text. See http://www.rebol.com/docs/dictionary.html for a list of additional series functions. For more information and examples, *be sure to read sections 6 and 7 from the REBOL/Core Users Guide* by Carl Sassenrath.

6.4.2 REBOL Strings

In REBOL, a "string" is simply a series of characters. If you have experience with other programming languages, this can be one of the sticking points in learning REBOL. REBOL's solution is actually a very powerful, easy to learn and consistent with the way other operations work in the language. Proper string management simply requires a good understanding of series. Take a look at the following examples to see how to do a few common operations:

```
the-string: "abcdefghijklmnopqrstuvwxyz"

; Left String:  (get the left 7 characters of the string):

copy/part the-string 7

; Right String:  (Get the right 7 characters of the string):

copy at tail the-string -7

; Mid String 1:  (get 7 characters from the middle of the string,
; starting with the 12th character):

copy/part (at the-string 12) 7

; Mid String 2:  (get 7 characters from the middle of the string,
; starting 7 characters back from the letter "m"):

copy/part (find the-string "m") -7

; Mid String 3:  (get 7 characters from the middle of the string,
; starting 12 characters back from the letter "t"):

copy/part (skip (find the-string "t") -12) 7

; 3 different ways to get just the 7th character:

the-string/7
pick the-string 7
seventh the-string

; Change "cde" to "123"

replace the-string "cde" "123"

; Several ways to change the 7th character to "7"

change (at the-string 7) "7"
poke the-string 7 #"7"  ; the pound symbol refers to a single character
poke the-string 7 (to-char "7")  ; another way to use single characters
print the-string

; Remove 15 characters, starting at the 3rd position:

remove/part (at the-string 3) 15
print the-string
```

```
; Insert 15 characters, starting at the 3rd position:

insert (at the-string 3) "cdefghijklmnopq"
print the-string

; Insert 3 instances of "-+" at the beginning of the string:

insert/dup head the-string "-+ " 3
print the-string

; Replace every instance of "-+ " with " ":

replace/all the-string "-+ "  " "
print the-string

; Remove spaces from a string (type "? trim" to see all its refinements!):

trim the-string
print the-string

; Get every third character from the string:

extract the-string 3

; Get the ASCII value for "c" (ASCII 99):

to-integer third the-string

; Get the character for ASCII 99 ("c"):

to-char 99

; Convert the above character value to a string value:

to-string to-char 99

; Convert any value to a string:

to-string now
to-string $2344.44
to-string to-char 99
to-string system/locale/months

; An even better way to convert values to strings:

form now
form $2344.44
form to-char 99
form system/locale/months  ; convert blocks to nicely formed strings

; Covert strings to a block of characters:

the-block: copy []
foreach item the-string [append the-block item]
probe the-block
```

REBOL's series functions are very versatile. Often, you can devise several ways to do the same thing:

```
; Remove the last part of a URL:
```

```
the-url: "http://website.com/path"
clear at the-url (index? find/last the-url "/")
print the-url

; Another way to do it:

the-url: "http://website.com/path"
print copy/part the-url (length? the-url)-(length? find/last the-url "/")
```

(Of course, REBOL has a built-in helper function to accomplish the above goal, directly with URLs):

```
the-url: http://website.com/path
print first split-path the-url
```

There are a number of additional functions that can be used to work specifically with string series. Run the following script for an introduction:

```
string-funcs: [
    build-tag checksum clean-path compress debase decode-cgi decompress
    dehex detab dirize enbase entab import-email lowercase mold parse-xml
    reform rejoin remold split-path suffix? uppercase
]
echo %string-help.txt  ; "echo" saves console activity to a file
foreach word string-funcs [
    print "_____________________________________________________________^/"
    print rejoin ["word:  " uppercase to-string word]  print ""
    do compose [help (to-word word)]
]
echo off
editor at read %string-help.txt 4
```

See http://www.rebol.com/docs/dictionary.html and http://rebol.com/docs/core23/rebolcore-8.html for more information about the above functions.

6.4.3 Indentation

Blocks often contain other blocks. Such compound blocks are typically indented with consecutive tab stops. *Starting and ending brackets are normally placed at the same indentation level*. This is conventional in most programming languages, because it makes complex code easier to read, by grouping things visually. For example, the compound block below:

```
big-block: [[may june july] [[1 2 3] [[yes no] [monday tuesday friday]]]]
```

can be written as follows to show the beginnings and endings of blocks more clearly:

```
big-block: [
    [may june july]
    [
        [1 2 3]
        [
            [yes no]
            [monday tuesday friday]
        ]
    ]
]
```

```
probe first big-block
probe second big-block
probe first second big-block
probe second second big-block
probe first second second big-block
probe second second second big-block
```

Indentation is not required, but it's very helpful.

6.4.4 More About Why/How Blocks are Useful

IMPORTANT: In REBOL, blocks can contain mixed data of *ANY* type (text and binary items, embedded lists of items (other blocks), variables, etc.):

```
some-items: ["item1" "item2" "item3" "item4"]
an-image: load http://rebol.com/view/bay.jpg
append some-items an-image

; "some-items" now contains 4 text strings, and an image!

; You can save that entire block of data, INCUDING THE BINARY
; IMAGE data, to your hard drive as a SIMPLE TEXT FILE:

save/all %some-items.txt some-items

; to load it back and use it later:

some-items: load %some-items.txt
view layout [image fifth some-items]
```

Take a moment to examine the example above. REBOL's block structure works in a way that is dramatically easy to use compared to other languages and data management solutions (much more simply than most database systems). It's is a very flexible, simple, and powerful way to store data in code! The fact that blocks can hold all types of data using one simple syntactic structure is a fundamental reason it's easier to use than other programming languages and computing tools. You can save/load block code to the hard drive as a simple text file, send it in an email, display it in a GUI, compress it and transfer it to a web server to be downloaded by others, transfer it directly over a point-to-point network connection, or even convert it to XML, encrypt, and store parts of it in a secure multiuser database to be accessed by other programming languages, etc...

Remember, all programming, and computing in general, is essentially about storing, organizing, manipulating, and transferring data of some sort. REBOL makes working with all types of data very easy - *just put any number of pieces of data, of any type, in between two brackets, and that data is automatically searchable, sortable, storable, transferable, and otherwise usable in your programs*.

6.4.5 Evaluating Variables in Blocks: Compose, Reduce, Pick and More

You will often find that you want to refer to an item in a block by its index (position number), as in the earlier 'some-items' example:

```
view layout [image some-items/5]
```

You may not, however, always know the specific index number of the data item you want to access. For example, as you insert data items into a block, the index position of the last item changes (it increases). You can obtain the index number of the last item in a block simply by determining the number of items in the block (the position number of the last item in a block is always the same as the total number of items in the block). In the example below, that index number is assigned the

variable word "last-item":

```
last-item: length? some-items
```

Now you can use that variable to pick out the last item:

```
view layout [image (pick some-items last-item)]

; In our earlier example, with 5 items in the block, the
; line above evaluates the same as:

view layout [image (pick some-items 5)]
```

You can refer to other items by adding and subtracting index numbers:

```
alert pick some-items (last-item - 4)
```

There are several other ways to do the exact same thing in REBOL. The "compose" function allows variables in parentheses to be evaluated and inserted as if they'd been typed explicitly into a code block:

```
view layout compose [image some-items/(last-item)]

; The line above appears to the interpreter as if the following
; had been typed:

view layout [image some-items/5]
```

The "compose" function is *very useful* whenever you want to refer to data at variable index positions within a block. The "reduce" function can also be used to produce the same type of evaluation. Function words in a reduced block should begin with the tick (') symbol:

```
view layout reduce ['image some-items/(last-item)]
```

Another way to use variable values explicitly is with the ":" format below. This code evaluates the same as the previous two examples:

```
view layout [image some-items/:last-item]
```

Think of the colon format above as the opposite of setting a variable. As you've seen, the colon symbol placed *after* a variable word *sets* the word to equal some value. A colon symbol placed *before* a variable word *gets* the value assigned to the variable, and inserts that value into the code as if it had been typed explicitly.

You can use the "index?" and "find" functions to determine the index position(s) of any data you're searching for in a block:

```
index-num: index? (find some-items "item4")
```

Any of the previous 4 formats can be used to select the data at the determined variable position:

```
print pick some-items index-num
print compose [some-items/(index-num)]
print reduce [some-items/(index-num)]
; no function words are used in the block above, so no ticks are required
print some-items/:index-num
```

Here's an example that displays variable image data contained in a block, using a foreach loop. The "compose" function is used to include dynamically changeable data (image representations), as if that data had been typed directly into the code:

```
photo1: load http://rebol.com/view/bay.jpg
photo2: load http://rebol.com/view/demos/palms.jpg

; The REBOL interpreter sees the following line as if all the code
; representing the above images had been typed directly in the block:

photo-block: compose [(photo1) (photo2)]

foreach photo photo-block [view layout [image photo]]
```

Block concepts may seem a bit vague at this point. The practical application of block structures will be presented much more thoroughly, by example, throughout this tutorial, and clarification about the usefulness of blocks will come from seeing them in working code. For additional detailed information about using blocks and series functions see http://www.rebol.com/docs/core23/rebolcore-6.html.

6.5 Conditions

6.5.1 If

Conditions are used to manage program flow. The most basic conditional evaluation is "if":

```
if (this expression is true) [do this block of code]
; parentheses are not required
```

Math operators are typically used to perform conditional evaluations: = < > <> (equal, less-than, greater-than, not-equal):

```
if now/time > 12:00 [alert "It's after noon."]
```

Here's an example that gets a username and password from the user, tests that data using an "if" evaluation, and alerts the user if the response is correct:

```
userpass: request-pass/title "Type 'username' and 'password'"
if (userpass = ["username" "password"]) [alert "Welcome back!"]
```

6.5.2 Either

"Either" is an if/then/else evaluation that chooses between two blocks to evaluate, based on whether the given condition is true or false. Its syntax is:

```
either (condition) [
    block to perform if the condition is true
][
    block to perform if the condition is false
]
```

For example:

```
either now/time > 8:00am [
    alert "It's time to get up!"
][
    alert "You can keep on sleeping."
]

userpass: request-pass
either userpass = ["username" "password"] [
    alert "Welcome back!"
][
    alert "Incorrect user/password combination!"
]
```

6.5.3 Switch

The "switch" evaluation chooses between numerous functions to perform, based on multiple evaluations. Its syntax is:

```
switch/default (main value) [
```

```
    (value 1) [block to execute if value 1 = main value
    (value 2) [block to execute if value 2 = main value]
    (value 3) [block to execute if value 3 = main value]
    ; etc...
] [default block of code to execute if none of the values match]
```

You can compare as many values as you want against the main value, and run a block of code for each matching value:

```
favorite-day:  request-text/title "What's your favorite day of the week?"

switch/default favorite-day [
    "Monday"    [alert "Monday is the worst!  The work week begins..."]
    "Tuesday"   [alert "Tuesdays and Thursdays are both ok, I guess..."]
    "Wednesday" [alert "The hump day - the week is halfway over!"]
    "Thursday"  [alert "Tuesdays and Thursdays are both ok, I guess..."]
    "Friday"    [alert "Yay!  TGIF!"]
    "Saturday"  [alert "Of course, the weekend!"]
    "Sunday"    [alert "Of course, the weekend!"]
] [alert "You didn't type in the name of a day!"]
```

6.5.4 Multiple Conditions: "and", "or", "all", "any"

You can check for more than one condition to be true, using the "and", "or", "all", and "any" words:

```
; first set some initial values all to be true:

value1: value2: value3: true

; then set some additional values all to be false:

value4: value5: value6: false

; The following prints "both true", because both the first
; condition AND the second condition are true:

either ( (value1 = true) and (value2 = true) ) [
    print "both true"
] [
    print "not both true"
]

; The following prints "both not true", because the second
; condition is false:

either ( (value1 = true) and (value4 = true) ) [
    print "both true"
] [
    print "not both true"
]

; The following prints "either one OR the other is true"
; because the first condition is true:

either ( (value1 = true) or (value4 = true) ) [
    print "either one OR the other is true"
] [
    print "neither is true"
]
```

```
; The following prints "either one OR the other is true"
; because the second condition is true:

either ( (value4 = true) or (value1 = true) ) [
    print "either one OR the other is true"
] [
    print "neither is true"
]

; The following prints "either one OR the other is true"
; because both conditions are true:

either ( (value1 = true) or (value4 = true) ) [
    print "either one OR the other is true"
] [
    print "neither is true"
]

; The following prints "neither is true":

either ( (value4 = true) or (value5 = true) ) [
    print "either one OR the other is true"
] [
    print "neither is true"
]
```

For comparisons involving more items, you can use "any" and "all":

```
; The following lines both print "yes", because ALL comparisons are true.
; "All" is just shorthand for the multiple "and" evaluations:

if ((value1 = true) and (value2 = true) and (value3 = true)) [
    print "yes"
]

if all [value1 = true  value2 = true  value3 = true] [
     print "yes"
]

; The following lines both print "yes" because ANY ONE of the comparisons
; is true. "Any" is just shorthand for the multiple "or" evaluations:

if ((value1 = true) or (value4 = true) or (value5 = true)) [
    print "yes"
]

if any [value1 = true  value4 = true  value5 = true] [
     print "yes"
]
```

6.6 Loops

6.6.1 Forever

"Loop" structures provide programmatic ways to methodically repeat actions, manage program flow, and automate lengthy data processing activities. The "forever" function creates a simple repeating loop. Its syntax is:

```
forever [block of actions to repeat]
```

The following code uses a forever loop to continually check the time. It alerts the user when 60 seconds has passed. *Notice the "break" function, used to stop the loop*:

```
alarm-time: now/time + :00:60
forever [if now/time = alarm-time [alert "1 minute has passed" break]]
```

Here's a more interactive version using some info provided by the user. Notice how the forever loop, if evaluation, and alert arguments are indented to clarify the grouping of related parameters:

```
event-name: request-text/title "What do you want to be reminded of?"
seconds: to-integer request-text/title "Seconds to wait?"
alert rejoin [
    "It's now " now/time ", and you'll be alerted in "
    seconds " seconds."
]
alarm-time: now/time + seconds
forever [
    if now/time = alarm-time [
        alert rejoin [
            "It's now "alarm-time ", and " seconds
            " seconds have passed.  It's time for: " event-name
        ]
        break
    ]
]
```

Here's a forever loop that displays/updates the current time in a GUI:

```
view layout [
    timer: field
    button "Start" [
        forever [
            set-face timer now/time
            wait 1
        ]
    ]
]
```

6.6.2 Loop

The "loop" function allows you to repeatedly evaluate a block of code, a specified number of times:

```
loop 50 [print "REBOL is great!"]
```

6.6.3 Repeat

Like "loop", the "repeat" function allows you to repeatedly evaluate a block of code, a specified number of times. It *additionally* allows you to specify a counter variable which is automatically incremented each time through the loop:

```
repeat count 50 [print rejoin ["This is loop #: " count]]
```

The above code does the same thing as:

```
count: 0
loop 50 [
    count: count + 1
    print rejoin ["This is loop #: " count]
]
```

6.6.4 For

"For" loops allow you to control repetition patterns that involve consecutively changing values. You specify a start value, end value, incremental value, and a variable name to hold the current value during the loop. Here's the "for" loop syntax:

for {variable word to hold current value} {starting value} {ending value} {incremental value} [block of code to perform, which can use the current variable value]

For example:

```
for counter 1 10 1 [print counter]
; starts on 1 and counts to 10 by increments of 1

for counter 10 1 -1 [print counter]
; starts on 10 and counts backwards to 1 by increments of -1

for counter 10 100 10 [print counter]
; starts on 10 and counts to 100 by increments of 10

for counter 1 5 .5 [print counter]
; starts on 1 and counts to 5 by increments of .5

for timer 8:00 9:00 0:05 [print timer]
; starts at 8:00am and counts to 9:00am by increments of 5 minutes

for dimes $0.00 $1.00 $0.10 [print dimes]
; starts at 0 cents and counts to 1 dollar by increments of a dime

for date 1-dec-2005 25-jan-2006 8 [print date]
; starts at December 12, 2005 and counts to January 25, 2006
; and by increments of 8 days

for alphabet #"a" #"z" 1 [prin alphabet]
; starts at the character a and counts to z by increments of 1 letter
```

Notice that REBOL properly increments dates, money, time, etc.

This "for" loop displays the first 5 file names in the current folder on your hard drive:

```
files: read %.
```

```
for count 1 5 1 compose [print files/(count)]
```

Notice the "compose" word used in the for loop. "files/1" represents the first item in the file list, "files/2" represents the second, and so on. The first time though the loop, the code reads as if [print files/1] had been typed in manually, etc.

The following "for" loop displays all the files in the current folder:

```
files: read %.
filecount: length? files
for count 1 filecount 1 compose [print files/(count)]
```

6.6.5 Foreach (very important!)

The "foreach" function lets you easily loop through a block of data. Its syntax is:

foreach {variable name referring to each consecutive item in the given block} [given block] [block of functions to be executed upon each item in the given block, using the variable name to refer to each successive item]

This example prints the name of every file in the current directory on your hard drive:

```
folder: read %.
foreach file folder [print file]
```

This line reads and prints each successive message in a user's email box:

```
foreach mail (read pop://user:pass@website.com) [print mail]
```

Here's a slightly more complex foreach example:

```
; define a block of text items:

some-names: ["John" "Bill" "Tom" "Mike"]

; define a variable used to count items in the block:

count: 0

; go through each item in the block:

foreach name some-names [

    ; increase the counter variable by 1, for each item:

    count: count + 1

    ; print the count number, and the associated text item:

    print rejoin ["Item " count ": " name]
]
```

Here's an example in which an empty block is created and data is appended using a foreach loop. The data is then converted to a text string and displayed in a GUI:

```
; define a block of text items:

some-names: ["John" "Bill" "Tom" "Mike"]

; create another new, empty block:

data-block: copy []

; define a variable used to count items in the block:

count: 0

; go through each item in the block:

foreach name some-names [

    ; increase the counter variable by 1, for each item:

    count: count + 1

    ; for each item, add some rejoined text to the originally empty block:

    append data-block rejoin ["Item " count ": " name newline]
]

; convert the newly created block to a string, and show it in a
; GUI text area widget:

view layout [area (to-string data-block)]
```

You can select multiple values from a block during each iteration of a foreach loop, using the following format:

```
users:  [
    "John Smith" "123 Tomline Lane Forest Hills, NJ" "555-1234"
    "Paul Thompson" "234 Georgetown Pl. Peanut Grove, AL" "555-2345"
    "Jim Persee" "345 Pickles Pike Orange Grove, FL" "555-3456"
    "George Jones" "456 Topforge Court Mountain Creek, CO" ""
    "Tim Paulson" "" "555-5678"
]

; Use the following format to get 3 consecutive values from the above
; block, each time through the loop:

foreach [name address phone] users [
    print rejoin [
        "^/Name:     " name       ; gets 1 value from the block
        "^/Address:  " address    ; gets the next value from the block
        "^/Phone:    " phone      ; gets a third value from the block
    ]
]
```

You will use the foreach function ***very*** *often* in REBOL code. It will be demonstrated many times, by example, throughout this tutorial.

6.6.6 Forall and Forskip

"Forall" loops through a block, incrementing the marked index number of the series as it loops through:

```
some-names: ["John" "Bill" "Tom" "Mike"]

foreach name some-names [print index? some-names]  ; index doesn't change
forall some-names [print index? some-names]  ; index changes

foreach name some-names [print name]
forall some-names [print first some-names] ; same effect as line above
```

"Forskip" works like forall, but skips through the block, jumping a periodic number of elements on each loop:

```
some-names: ["John" "Bill" "Tom" "Mike"]
forskip some-names 2  [print first some-names]
```

6.6.7 While and Until

The "while" function repeatedly evaluates a block of code while the given condition is true. While loops are formatted as follows:

```
while [condition] [
    block of functions to be executed while the condition is true
]
```

This example counts to 5:

```
x: 1  ; create an initial counter value
while [x <= 5] [
    alert to-string x
    x: x + 1
]
```

In English, that code reads:

```
"x" initially equals 1.
While x is less than or equal to 5, display the value of x,
then add 1 to the value of x and repeat.
```

Some additional "while" loop examples:

```
while [not request "End the program now?"] [
    alert "Select YES to end the program."
]
; "not" reverses the value of data received from
; the user (i.e., yes becomes no and visa versa)

alert "Please select today's date"
while [request-date <> now/date] [
    alert rejoin ["Please select TODAY's date.  It's " now/date]
]

while [request-pass <> ["username" "password"]] [
    alert "The username is 'username' and the password is 'password'"
]
```

"Until" loops are similar to "while" loops. They do everything in a given block, repeatedly, *until* the last expression in the block evaluates to true:

```
x: 10
until [
    print rejoin ["Counting down: " x]
    x: x - 1
    x = 0
]
```

The example below uses several loops to alert the user to feed the cat, every 6 hours between 8am and 8pm. It uses a for loop to increment the times to be alerted, a while loop to continually compare the incremented times with the current time, and a forever loop to do the same thing every day, continuously. Notice the indentation:

```
forever [
    for timer 8:00am 8:00pm 6:00 [
        while [now/time <= timer] [wait :00:01]
        alert rejoin ["It's now " now/time ".  Time to feed the cat."]
    ]
]
```

6.7 User Defined Functions and Imported Code

REBOL's built-in functions satisfy many fundamental needs. To achieve more complex or specific computations, you can create your own function definitions.

Data and function words contained in blocks can be evaluated (their actions performed and their data values assigned) using the "do" word. **Because of this, *any* block of code can essentially be treated as a function**. That's a powerful key element of the REBOL language design:

```
some-actions: [
    alert "Here is one action."
    print "Here's a second action."
    write %/c/anotheraction.txt "Here's a third action."
]

do some-actions
```

New function words can also be defined using the "does" and "func" words. "Does" is included directly after a word label definition, and forces a block to be evaluated every time the word is encountered:

```
more-actions: does [
    alert "4"
    alert "5"
    alert "6"
]

; now to use that function, just type the word label:

more-actions
```

Here's a useful function to clear the command line screen in the REBOL interpreter.

```
cls: does [prin "^(1B)[J"]

cls
```

The "func" word creates an executable block in the same way as "does", but additionally allows you to pass your own specified parameters to the newly defined function word. The first block in a func definition contains the name(s) of the variable(s) to be passed. The second block contains the actions to be taken. Here's the "func" syntax:

```
func [names of variable(s) to be passed] [
    actions to be taken with those variables
]
```

This function definition:

```
sqr-add-var: func [num1 num2] [print square-root (num1 + num2)]
```

Can be used as follows. *Notice that no brackets, braces, or parentheses are required to contain the data arguments*. Data parameters simply follow the function word, on the same line of code:

```
sqr-add-var 12 4      ; prints "4",  the square root of 12 + 4  (16)
sqr-add-var 96 48     ; prints "12", the square root of 96 + 48 (144)
```

Here's a simple function to display images:

```
display: func [filename] [view layout [image load to-file filename]]

display (request-file)
```

By default, the last value evaluated by a function is returned when the function is complete:

```
concatenate: func [string1 string2] [join string1 string2]

string3: concatenate "Hello " "there."
print string3
```

By default, values used inside functions are treated as *global*, which means that if any variables are changed inside a function, they will be changed throughout the rest of your program:

```
x: 10

change-x-globally: func [y z] [x: y + z]

change-x-globally 10 20
print x
```

You can change this default behavior, and specify that any value be treated as *local* to the function (not changed throughout the rest of your program), by using the "/local" refinement:

```
x: 10

change-x-locally: func [y z /local x] [x: y + z]

change-x-locally 10 20     ; inside the function, x is now 30
print x                    ; outside the function, x is still 10
```

You can specify refinements to the way a function operates, simply by preceding optional operation arguments with a forward slash ("/"):

```
compute: func [x y /multiply /divide /subtract] [
    if multiply [return x * y]
    if divide   [return x / y]
    if subtract [return x - y]
    return x + y
]

compute/multiply 10 20
compute/divide 10 20
compute/subtract 10 20
compute 10 20
```

The "help" function provides usage information for any function, including user defined functions:

```
help for
help compute
```

You can include documentation for any user defined function by including a text string as the first item in it's argument list. This text is included in the description displayed by the help function:

```
doc-demo: func ["This function demonstrates doc strings"] [help doc-demo]
doc-demo
```

Acceptable data types for any parameter can be listed in a block, and doc strings can also be included immediately after any parameter:

```
concatenate-string-or-num: func [
    "This function will only concatenate strings or integers."
    val1 [string! integer!] "First string or integer"
    val2 [string! integer!] "Second string or integer"
] [
    join val1 val2
]

help concatenate-string-or-num
concatenate-string-or-num "Hello " "there."  ; this works correctly
concatenate-string-or-num 10 20              ; this works correctly
concatenate-string-or-num 10.1 20.3          ; this creates an error
```

6.7.1 Importing Code

You can "do" a module of code contained in any text file, *as long as it contains the minimum header **"REBOL [ ]"*** (this includes HTML files and any other files that can be read via REBOL's built-in protocols). For example, if you save the previous functions in a text file called "myfunctions.r":

```
REBOL []  ; THIS HEADER TEXT MUST BE INCLUDED AT THE TOP OF ANY REBOL FILE

sqr-add-var: func [num1 num2] [print square-root (num1 + num2)]
display: func [filename] [view layout [image load filename]]
cls: does [prin "^(1B)[J"]
```

You can import and use them in your current code, as follows:

```
do %myfunctions.r

; now you can use those functions just as you would any other
; native function:

sqr-add-var
display
cls
```

Here's an example function that plays a .wave sound file. Save this code as C:\play_sound.r:

```
REBOL [title: "play-sound"]  ; you can add a title to the header

play-sound: func [sound-file] [
```

```
    wait 0
    ring: load sound-file
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
]
```

Then run the code below to import the function and play selected .wav files:

```
do %/c/play_sound.r

play-sound %/C/WINDOWS/Media/chimes.wav
play-sound to-file request-file/file %/C/WINDOWS/Media/tada.wav
```

Imported files can contain data definitions and any other executable code, including that which is contained in additional nested source files imported with the "do" function. Any code or data contained in a source file is evaluated when the file is "do"ne.

6.8 Quick Review and Synopsis

The list below summarizes some key characteristics of the REBOL language. Knowing how to put these elements to use constitutes a fundamental understanding of how REBOL works:

1. To start off, REBOL has hundreds of built-in function words that perform common tasks. As in other languages, function words are typically followed by passed parameters. Unlike other languages, passed parameters are placed immediately after the function word and are *not* necessarily enclosed in parenthesis. To accomplish a desired goal, functions are arranged in succession, one after another. The value(s) returned by one function are often used as the argument(s) input to another function. Line terminators are not required at any point, and all expressions are evaluated in left to right order, then vertically down through the code. Empty white space (spaces, tabs, newlines, etc.) can be inserted as desired to make code more readable. Text after a semicolon and before a new line is treated as a comment. You can complete significant work by simply knowing the predefined functions in the language, and organizing them into a useful order.
2. REBOL contains a rich set of conditional and looping structures, which can be used to manage program flow and data processing activities. If, switch, while, for, foreach, and other typical structures are supported.
3. Because many common types of data values are automatically recognized and handled natively by REBOL, calculating, looping, and making conditional decisions based upon data content is straightforward and natural to perform, without any external modules or toolkits. Numbers, text strings, money values, times, tuples, URLs, binary representations of images, sounds, etc. are all automatically handled. REBOL can increment, compare, and perform proper computations on most common types of data (i.e., the interpreter automatically knows that 5:32am + 00:35:15 = 6:07:15am, and it can automatically apply visual effects to raw binary image data, etc.). Network resources and Internet protocols (http documents, ftp directories, email accounts, dns services, etc.) can also be accessed natively, just as easily as local files. Data of any type can be written to and read from virtually any connected device or resource (i.e., "write %file.txt data" works just as easily as "write ftp://user:pass@website.com data", using the same common syntax). The percent symbol ("%") and the syntax "%(/drive)/path/path/.../file.ext" are used cross-platform to refer to local file values on any operating system.
4. Any data or code can be assigned a word label. The colon character (":") is used to assign word labels to constants, variable values, evaluated expressions, functions, and data/action blocks of any type. Once assigned, variable words can be used to represent all of the data and/or actions contained in the given expression, block, etc. Just put a colon at the end of a word, and thereafter it represents all the following actions and/or data. That forms a significant part of the REBOL language structure, and is the basis for it's flexible natural language dialecting abilities.
5. Multiple pieces of data are stored in "blocks", which are delineated by starting and ending brackets ("[]"). Blocks can contain data of *any* type: groups of text strings, arrays of binary data, collections of actions (functions), other enclosed blocks, etc. Data items contained in blocks are separated by white space. Blocks can be automatically treated as lists of data, called "series", and manipulated using built-in functions that enable searching, sorting, ordering, and otherwise organizing the blocked data. Data and function words contained in blocks can be evaluated (their actions performed and their data values assigned) using the "do" word. New function words can also be defined using the "does" and "func" words. "Does" forces a block to be evaluated every time its word label is encountered. The "func" word creates an executable block in the same way as "does", but additionally allows you to pass your own specified parameters to the newly defined function word. You can "do" a module of code contained in a text file, as long as it contains the minimum header "rebol[]". Blocks are also used to delineate most of the syntactic structures in REBOL (i.e., in conditional evaluations, function definitions, etc.).
6. The syntax "view layout [block]" is used to create basic GUI layouts. You can add graphic widgets to the layout simply by adding widget identifier words to the enclosed block: "button", "field", "text-list", etc. Color, position, spacing, and other facet words can be added after each widget identifier. Action blocks added immediately after any widget will perform the enclosed functions whenever the widget is activated (i.e., when the widget is clicked with a mouse, when the enter key pressed, etc.). Path refinements can be used to refer to items in the GUI layout (i.e., "face/offset" refers to the position of the selected widget face). Those simple guidelines can be used to create useful GUIs for data input and output, in a way that's native (doesn't require any external toolkits) and much easier than any other language.

6.9 A Telling Comparison

To provide a quick idea of how much easier REBOL is than other languages, here's a short example. The following code to create a basic program window with REBOL was presented earlier:

```
view layout [size 400x300]
```

It works on every type of computer, in exactly the same way.

Code for the same simple example is presented below in the popular programming language "C++". It does the exact same thing as the REBOL one-liner above, except it only works in Microsoft Windows. If you want to do the same thing on a Macintosh computer, you need to memorize a completely different page of C++ code. The same is true for Unix, Linux, Beos, or any other operating system. You have to learn enormous chunks of code to do very simple things, and those chunks of code are different for every type of computer. Furthermore, you typically need to spend a semester's worth of time learning very basic things about coding format and fundamentals about how a computer 'thinks' before you even begin to tackle useful basics like the code below:

```
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);

/*  Make the class name into a global variable  */
char szClassName[ ] = "C_Example";

int WINAPI
WinMain (HINSTANCE hThisInstance,
         HINSTANCE hPrevInstance,
         LPSTR lpszArgument,
         int nFunsterStil)

{
    HWND hwnd;
    /* This is the handle for our window */
    MSG messages;
    /* Here messages to the application are saved */
    WNDCLASSEX wincl;
    /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;
    /* This function is called by windows */
    wincl.style = CS_DBLCLKS;
    /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;
    /* No menu */
    wincl.cbClsExtra = 0;
    /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;
    /* structure or the window instance */
```

```
    /* Use Windows's default color as window background */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register window class. If it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,
            /* Extended possibilites for variation */
           szClassName,
            /* Classname */
           "C_Example",
            /* Title Text */
           WS_OVERLAPPEDWINDOW,
            /* default window */
           CW_USEDEFAULT,
            /* Windows decides the position */
           CW_USEDEFAULT,
            /* where the window ends up on the screen */
           400,
            /* The programs width */
           300,
            /* and height in pixels */
           HWND_DESKTOP,
            /* The window is a child-window to desktop */
           NULL,
            /* No menu */
           hThisInstance,
            /* Program Instance handler */
           NULL
            /* No Window Creation data */
           );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

    /* Run the message loop.
        It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages
            into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 -
        The value that PostQuitMessage() gave */
    return messages.wParam;
}

/*  This function is called by the Windows
        function DispatchMessage()  */

LRESULT CALLBACK
WindowProcedure (HWND hwnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    switch (message)
    /* handle the messages */
```

```
    {
        case WM_DESTROY:
            PostQuitMessage (0);
                /* send a WM_QUIT to the message queue */
            break;
        default:
            /* for messages that we don't deal with */
            return DefWindowProc (hwnd, message,
                wParam, lParam);
    }

    return 0;
}
```

Yuck. Back to REBOL...

7. More Essential Topics

7.1 Built-In Help and Online Resources

The "help" function displays required syntax for any REBOL function:

```
help print
```

"?" is a synonym for "help":

```
? print
```

The "what" function lists all built-in words:

```
what
```

Together, those two words provide a built-in reference guide for the entire core REBOL language. Here's a script that saves all the above documentation to a file. Give it a few seconds to run:

```
echo %words.txt what echo off   ; "echo" saves console activity to a file
echo %help.txt
foreach line read/lines %words.txt [
    word: first to-block line
    print "_____________________________________________________________^/"
    print rejoin ["word:  " uppercase to-string word]  print ""
    do compose [help (to-word word)]
]
echo off
editor at read %help.txt 4
```

You can use help to search for defined words and values, when you can't remember the exact spelling of the word. Just type a portion of the word (hitting the *tab* key will also show a list of words for automatic *word completion*):

```
? to-          ; shows a list of all built-in type conversions
? reques       ; shows a list of built-in requester functions
? "load"       ; shows all words containing the characters "load"
? "?"          ; shows all words containing the character "?"
```

Here are some more examples of ways to search for useful info using help:

```
? datatype!    ; shows a list of built-in data types
? function!    ; shows a list of built-in functions
? native!      ; shows a list of native (compiled C code) functions
? char!        ; shows a list of built-in control characters
? tuple!       ; shows a list of built-in colors (RGB tuples)
? .gif         ; shows a list of built-in .gif images
```

You can view the *source code* for built-in "mezzanine" (non-native) functions with the "source" function. There is a *huge volume* of REBOL code accessible right in the interpreter, and all of the mezzanine functions were created by the language's designer, Carl Sassenrath. Studying mezzanine source is a great way to learn more about advanced REBOL code patterns:

```
source help
source request-text
source view
source layout
source ctx-viewtop  ; try this:  view layout [image load ctx-viewtop/13]
```

7.1.1 The REBOL System Object, and Help with GUI Widgets

"Help system" displays the contents of the REBOL system object, which contains many important settings and values. You can explore each level of the system object using path notation, like this:

```
? system/console/history        ; the current console session history
? system/options
? system/locale/months
? system/network/host-address
```

You can find info about all of REBOL's GUI components in "system/view/VID":

```
? system/view/VID
```

The system/view/VID block is so important, REBOL has a built-in short cut to refer to it:

```
? svv
```

You'll find a list of REBOL's GUI widgets in "svv/vid-styles". Use REBOL's "editor" function to view large system sections like this:

```
editor svv/vid-styles
```

Here's a script that neatly displays all the words in the above "svv/vid-styles" block:

```
foreach i svv/vid-styles [if (type? i) = word! [print i]]
```

Here's a more concise way to display the above widgets, using the "extract" function:

```
probe extract svv/vid-styles 2
```

This script lets you browse the object structure of each widget:

```
view layout [
    text-list data (extract svv/vid-styles 2) [
        a/text: select svv/vid-styles value
        show a focus a
    ]
    a: area 500x250
]
```

REBOL's GUI layout words are available in "svv/vid-words":

```
? svv/vid-words
```

The following script displays all the images in the svv/image-stock block:

```
b: copy []
foreach i svv/image-stock [if (type? i) = image! [append b i]]
v: copy [] foreach i b [append v reduce ['image i]]
view layout v
```

The changeable attributes ("facets") available to all GUI widgets are listed in "svv/facet-words":

```
editor svv/facet-words
```

Here's a script that neatly displays all the above facet words:

```
b: copy []
foreach i svv/facet-words [if (not function? :i) [append b to-string i]]
view layout [text-list data b]
```

Some GUI widgets have additional facet words available. The following script displays all such functions, and their extra attributes:

```
foreach i (extract svv/vid-styles 2) [
    x: select svv/vid-styles i
    ; additional facets are held in a "words" block:
    if x/words [
        prin join i ": "
        foreach q x/words [
            if not (function? :q) [prin join q " "]
        ]
        print ""
    ]
]
```

To examine the function(s) that handle any of the additional facets for the widgets above, type the path to the widget's "words" block, i.e.:

```
svv/vid-styles/TEXT-LIST/words
```

For more information on system/view/VID, see http://www.mail-archive.com/rebol-bounce@rebol.com/msg01898.html and http://www.rebol.org/ml-display-message.r?m=rmlHJNC.

It's important to note that you can SET any system value. Just use a colon, like when assigning variable values:

```
system/user/email: user@website.com
```

Familiarity with the system object yields many useful tools.

7.1.2 Viewtop Resources

The REBOL desktop that appears by default when you run the view.exe interpreter can be used as a

gateway into a world of "Rebsites" that developers use to share useful code. Surfing the public rebsites is a great way to explore the language more deeply. All of the code in the rebol.org archive, and much more, is available on the rebsites. When typing at the interpreter console, the "desktop" function brings up the REBOL desktop (also called the "Viewtop"):

```
desktop
```

Click the "REBOL" or "Public" folders to see hundreds of interesting demos and useful examples. Source code for every example is available by right-clicking individual program icons and selecting "edit". You don't need a web browser or any other software to view the contents of Rebsites - the Viewtop and all its features are part of the REBOL executable. You can learn volumes about the REBOL language using *only* the resources built directly into the 600k interpreter!

For detailed, categorized, and cross-referenced information about built-in functions, see the REBOL Dictionary rebsite, found in the REBOL desktop folder REBOL->Tools (an HTML version is also available at http://www.rebol.com/docs/dictionary.html).

7.1.3 Online Documentation, The Mailing List and The AltME Community Forum

If you can't find answers to your REBOL programming questions using built-in help and resources, the first place to look is http://rebol.com/docs.html. Googling online documentation also tends to provide quick results, since the word "REBOL" is uncommon.

To ask a question directly of other REBOL developers, you can join the community mailing list by sending an email to *rebol-request@rebol.com* , with the word *"subscribe"* in the subject line. Use your normal email program, or just paste the following code into your REBOL interpreter (be sure your email account settings are set up correctly in REBOL):

```
send rebol-request@rebol.com "subscribe"
```

You can also ask questions of numerous gurus and regular users in **AltME**, a messaging program which makes up the most active forum of REBOL users around the world. Rebol.org maintains a searchable history of several hundred thousand posts from both the mailing list and AltME, along with a rich script archive. The REBOL user community is friendly, knowledgeable and helpful, and you will typically find answers to just about any question already in the archives. Unlike other programming communities, *REBOL does not have a popular web based support forum*. AltME is the primary way that REBOL developers interact. If you want to speak with others, you must download the AltME program and set up a user account (it's fast and easy to do). Just follow the instructions at http://www.rebol.org/aga-join.r.

7.2 Saving and Running REBOL Scripts

So far in this tutorial, you've been typing or copying/pasting code snippets directly into the REBOL interpreter. As you begin to work with longer examples and full programs, you'll need to save your scripts for later execution. *Whenever you save a REBOL program to a text file, the code must begin with the following bit of header text*:

```
REBOL []
```

That header tells the REBOL interpreter that the file contains a valid REBOL program. You can optionally document any information about the program in the header block. The "title" variable in the header block is displayed in the title bar of GUI program windows:

```
REBOL [
    title:  "My Program"
    author: "Nick Antonaccio"
    date:   29-sep-2009
]
view layout [text 400 center "Look at the title bar."]
```

The code below is a web cam video viewer program. Type in or copy/paste the complete code source below into a text editor such as Windows Notepad or REBOL's built-in text editor (type "editor none" at the REBOL console prompt). Save the text as a file called "webcam.r" on your C:\ drive.

```
REBOL [title: "Webcam Viewer"]

; try http://www.webcam-index.com/USA/ for more webcam links.

temp-url: "http://209.165.153.2/axis-cgi/jpg/image.cgi"
while [true]  [
    webcam-url: to-url request-text/title/default "Web cam URL:" temp-url
    either attempt [webcam: load webcam-url] [
        break
    ] [
        either request [
            "That webcam is not currently available." "Try Again" "Quit"
        ] [
            temp-url: to-string webcam-url
        ] [
            quit
        ]
    ]
]
resize-screen: func [size] [
    webcam/size: to-pair size
    window/size: (to-pair size) + 40x72
    show window
]
window: layout [
    across
    btn "Stop" [webcam/rate: none show webcam]
    btn "Start" [
        webcam/rate: 0
        webcam/image: load webcam-url
        show webcam
    ]
    rotary "320x240" "640x480" "160x120" [
```

```
        resize-screen to-pair value
    ]
    btn "Exit" [quit] return
    webcam: image load webcam-url  320x240
    with [
        rate: 0
        feel/engage: func [face action event][
            switch action [
                time [face/image: load webcam-url show face]
            ]
        ]
    ]
]
view center-face window
```

Once you've saved the webcam.r program to C:\, you can run it in any one of the following ways:

1. ***If you've already installed REBOL on your computer, just double-click your saved ".r" script file*** (find the C:\webcam.r file icon in your file explorer (click My Computer -> C: -> webcam.r)). By default, during REBOL's initial installation, all files with a ".r" extension are associated with the interpreter. They can be clicked and run *as if they're executable programs, **just like ".exe" files***. The REBOL interpreter automatically opens and executes any selected ".r" text file. This is the most common way to run REBOL scripts, and it works the same way on all major graphic operating systems. If you want other people to be able to run your scripts, just have them download and install the tiny REBOL interpreter - it only takes a few seconds.
2. Use the built-in editor in REBOL. Type "editor %/c/webcam.r" at the interpreter prompt, or type "editor none" and copy/paste the script into the editor. *Pressing F5 in the editor will automatically save and run the script*. This is a convenient way to work with scripts, and enables REBOL to be its own simple, self contained IDE.
3. Type "do %/c/webcam.r" into the REBOL interpreter.
4. Scripts can be run at the command line. In Windows, copy rebol.exe and webcam.r to the same folder (C:\), then click Start -> Run, and type "C:\rebol.exe C:\webcam.r" (or open a DOS box and type the same thing). Those commands will start the REBOL interpreter and do the webcam.r code. You can also create a text file called webcam.bat, containing the text "C:\rebol.exe C:\webcam.r" . Click on the webcam.bat file in Windows, and it'll run those commands. In Unix, you can also run scripts at scheduled times with Cron. Just enter the path to the script.
5. Use a program such as XpackerX to package and distribute the program. XpackerX allows you to wrap the REBOL interpreter and webcam.r program into a single executable file that has a clickable icon, and automatically runs both files. That allows you to create a single file executable Windows program that can be distributed and run like any other application. Just click it and run... (this technique is covered in the next section).
6. Buy the commercial "SDK" version of REBOL, which provides the most secure method of packaging REBOL applications.

VERY IMPORTANT: To turn off the default security requester that continually asks permission to read/write the hard drive, type "secure none" in the REBOL interpreter, and then run the program with "do {filename}". Running *"C:\rebol.exe -s {filename}"* does the same thing . The "-s" launches the REBOL interpreter without any security features turned on, making it behave like a typical Windows program.

7.3 "Compiling" REBOL Programs - Distributing Packaged .EXE Files

The REBOL.exe interpreter is tiny and does not require any installation to operate properly. By packaging it, your REBOL script(s), and any supporting data file(s) into a single executable with an icon of your choice, XpackerX works like a REBOL 'compiler' that produces regular Windows programs that look and act just like those created by other compiled languages. To do that, you'll need to create a text file in the following format (save it as "template.xml"):

```
<?xml version="1.0"?>
<xpackerdefinition>
    <general>
        <!--shown in taskbar -->
        <appname>your_program_name</appname>
        <exepath>your_program_name.exe</exepath>
        <showextractioninfo>false</showextractioninfo>
        <!-- <iconpath>c:\icon.ico</iconpath> -->
    </general>
    <files>
        <file>
            <source>your_rebol_script.r</source>
            <destination>your_rebol_script.r</destination>
        </file>
        <file>
            <source>C:\Program Files\rebol\view\Rebol.exe</source>
            <destination>rebol.exe</destination>
        </file>
        <!--put any other data files here -->
    </files>
    <!-- $FINDEXE, $TMPRUN, $WINDIR, $PROGRAMDIR, $WINSYSDIR -->
    <onrun>$TMPRUN\rebol.exe -si $TMPRUN\your_rebol_script.r</onrun>
</xpackerdefinition>
```

Just download the free XpackerX program and alter the above template so that it contains the filenames you've given to your script(s) and file(s), and the correct path to your REBOL interpreter. Run XpackerX, and it'll spit out a beautifully packaged .exe file that requires no installation. Your users do *not* need to have REBOL installed to run this type of executable. To them it appears and runs just like any other native compiled Windows program. What actually happens is that every time your packaged .exe file runs, the REBOL interpreter and your script(s)/data file(s) are unzipped into a temporary folder on your computer. When your script is done running, the temporary folder is deleted.

Most modern compression (zip) applications have an "sfx" feature that allows you to create .exe packages from zip files. You can create a packaged REBOL .exe in the same way as XpackerX using just about any sfx packaging application (there are dozens of freeware zip/compression applications that can do this - use the one you're most familiar with). There is an explanation of how to use the NSIS install creator to make REBOL .exe's here.

To create a self-extracting REBOL executable for Linux, first create a .tgz file containing all the files you want to distribute (the REBOL interpreter, your script(s), any external binary files, etc.). For the purposes of this example, name that bundle "rebol_files.tgz". Next, create a text file containing the following code, and save it as "sh_commands":

```
#!/bin/sh
SKIP=`awk '/^__REBOL_ARCHIVE__/ { print NR + 1; exit 0; }' $0`
tail +$SKIP $0 | tar xz
exit 0
__REBOL_ARCHIVE__
```

Finally, use the following command to combine the above script file with the bundled .tgz file:

```
cat sh_commands rebol_files.tgz > rebol_program.sh
```

The above line will create a single executable file named "rebol_program.sh" that can be distributed and run by end users. The user will have to set the file permissions for rebol_program.sh to executable before running it ("chmod +x rebol_program.sh"), or execute it using the syntax "sh rebol_program.sh".

7.4 Embedding Binary Resources and Using REBOL's Built In Compression

The following program can be used to encode external files (images, sounds, DLLs, .exe files, etc.) so that they can be included within the *text* of your program code. Use "load (data)" to make use of any text data created by this program:

```
REBOL [Title: "Simple Binary Embedder"]

system/options/binary-base: 64
file: to-file request-file/only
data: read/binary file
editor data
```

The example below uses a text representation of the image at http://musiclessonz.com/test.png, *encoded with the program above*:

```
picture: load 64#{
iVBORw0KGgoAAAANSUhEUgAAAFUAAABkCAIAAAB4sesFAAAAE3RFWHRTb2Z0d2Fy
ZQBSRUJPTC9WaWV3j9kWeAAAAU1JREFUeJztlzEOgzAQBHkaT7s2ryZUUZoYRz4t
e9xsSzTjEXIktqP3trsPcPPo7z36e4/+3qO/9y76766jBb86nUyZP
lM7kidKZPFE6kydq/Pjxq/nSElGv3qv50vj/O436l+hNQM6Z93+P3zqefAw12Fyqh
v/ToX+4Pt0ubiNKZPFE6Ux5q/O/436lkh6affvrpp38tPPqX99Ps
/meidCZPlM7kidKZPFE6kydKZ/JE6UyeaJ0Jk+n8LL
j/7l/nC7tIkonekLdXm9dafSmeinn376D/rpp5/vv1GqBkT37+FR/9yf7hd2kSU
zuSJ0pk8UTqTpn8kTpTJ4onckTpTN5onQmT5TO5InSmTxRROpMnasbE92/h0b/Q
//jR33v09x79vUd/73XvfwNmVzlr+eOLmgAAAABJRU5ErkJggg==
}
view layout [image picture]
```

The program below allows you to *compress* and embed files in your code. **This program will be referred to many times throughout the tutorial**. Save it to a .r file so that it can be run later:

```
REBOL [Title: "Binary Resource Embedder"]

system/options/binary-base: 64
file: to-file request-file/only
if not file [quit]
uncompressed: read/binary file
compressed: compress to-string uncompressed
editor compressed
alert rejoin ["Uncompressed size:  " length? uncompressed
    " bytes.  Compressed size: " length? compressed " bytes."]
```

To use the compressed version of data created by the program above, use the following code:

```
to-binary decompress {compressed data}
```

For example:

```
image-compressed: load to-binary decompress 64#{
eJzrDPBz5+WS4mJgYOD19HAJAtL/GRgYdTiYgKzm7Z9WACnhEteIkuD8tJLyxKJU
hiBXJ38f/bDM1PL+m2IVDAzsFz1dHEMq5ry9u3GijKcAy0Fh3kVzn/0XmRW5WXGV
sUF25EOmKwrSjrrF9v89o//u+cs/IS75763Tv7ZO/5qt//p63LX1e9fEV0fu/7ap
7m0qZRIJf+2DmGZoVER5MQiz+ntzJix6kKnJ6CNio6va0Nm0fCmLQeCHLVMY1Ljm
```

```
TRM64HLwMpGK/334Hf4n+vkn+1pr9md7jAVsYv+X8Z3Z+M/yscIX/j32H7sl/0j3
KK+of/CX8/X63sV1w51WqNj1763MjOS/xcccX8hzzFtXDwyXL9f/P19/f0vxz4f2
OucaHfmZDwID+P7Hso/5snw8m+qevH1030pG4kr8fhNC4f/34Z89ov+vHe4vAeut
SsdqX8T/OYUCv9iblr++f67R8pp9ukzLv8YHL39tL07o+3pekn1h/dDVBgzLU/d3
9te/Lki4cNgBmA6/lO+J/RPdzty8Rr5y94/tfOxsX6/r8xJK0/UW9vlH93/9oAzR
e09yKIUBVbT9/br/U/m7x6CU98VAAJS2ZPPF/197eEDhtfs9vX9rDzc6/v3qzUyo
nJA/dz76Y77tHw+w3gXlbEMpDKihza/+7/o/c3+DU54tDwsobR2/fXR/qYXBiV8T
t3eDEmpA/d9LDASK0y/tnz+H/Ynmt78E1vti7lAKA6pouxz/X7v+uR045ZFdRE6x
1q21pG7NiSzx1f5R40pvvdNn+oB1P4Onq5/LOqeEJgCemFy1KQgAAA==
}
view layout [image image-compressed]
```

7.5 Running Command Line Applications

The "call" function executes commands in your computer's operating system (i.e., DOS and Unix commands). The example below opens Windows' Notepad to edit the "C:\YOURNAME.txt" text file created earlier (leaving out the /show option runs the program in a hidden window):

```
call/show "notepad.exe c:\YOURNAME.txt"
```

This next example opens Windows' Paint program to edit an image we downloaded earlier in the tutorial:

```
call/show "mspaint.exe c:\bay.jpg"
```

Here's an example that embeds an executable program into the code, decompresses, and writes the program to the hard drive, and then runs it with the call function:

```
program: load to-binary decompress 64#{
eJztF11sU2X03K4VqJsrkZJp6OzchhFJsx8qDB9od1fHdIO6ds7AgJX2jttyey/p
vWUjJuNnmNhMibzwaCSLi+EBE1ziGIkBGh0BSYTwwAMme9Dk4kgkgSiKcj3nu7es
QrKFhMUQOcn5+c7fd875+vXe27FJAg4AbIiGAQwWIwZMEbqTcmODN5xRdmRi6aoy
Z83YogngLlaNtV+s6kV7q9KelHeu9LYqQTXt7e/v97UqLcLuqKJIvriShnAIoJ0r
gXvPn+StlDAF5dyzHLwAdlw4TZ1Mm7oQvWDu7jKLslsxBc4KQ30bb9bMHF3F/D5j
MFAHEIbHD+cwb88s9riSEIjvK7EKogZs//bxAvQmYlqM5JsOUwHPWFgEAYDTvqTp
eYdy1Fn5Sh/O96h9nLrrDcD4IpQm7UOkWL/nt6MlqMvxrkl+GVWS7xqWalzDzqGz
9rbyD5ehpmnl+ezt3M/RSPe7Q9/ajeh5+9Ztm3vKh9xoM7SaimLUR18C2JKf+Kg2
APoJwzDOuiAF+hHU/pHXryObdLyP+y2kEhx7UaLfo0gq/RJa60/n88Ndrpz7FmqG
u5bk3L8zwdWXc0+jdOYXkn4lnYfW++/qOPLyDz7BfH3jTXVnplx949inhPvnSgw/
8RSIHM7P8PdSUYtxlxSkONE+o/u7EkNElMbpcuRKUhTjmLH/iHbDQQ7DHqL77zbh
oQxeRa9duBQHkRj+HnIdr7y/e178AvmmnHt5VQAmaNo59/EZ8QSJAY7EURJvMu2x
KipYj2CaEToYve2eYYiwl4rWY6jN8RWF5XtsuWSyhO7aJG8XXQFkNdWYIqIHK8nH
8FOSFJMoteEfZfQEo1SNCPCW2/BTjWK1uXkp9dDDegjrDqpkAUtiJhNp4ma3qUrx
MG6dqkyFMQ2ExQmaxgU2c/07D2ZJsCz3Q68Xh76Cvac2pZwi8jCO8rIZd4jielmc
uHxmsEMe1vMBZJf0YY8Pda95yH5p+tWrI86XMZbTE5a1gVlXFKyryeowp0Cy4Wf+
hdSrWGp26N008hW4XnS6/OBS7MnUVHoK0osoTV+22qF56c95qKdtZBzB66J/imSc
/Rmsg/KDdHFbA9O3RrZWByD/qPf1KTCwze3y2KCbn9vnP4ExoItiwr11zvncqq6+
oXGV//XVa5qCzXxL6M3ZfBfMZyFPBvywgD3FGDjLnGVl83o4T+HJAZ/PFxWTqrcj
GxerHljRqyL9sWXxqU2/nkHki1H4HDkvJeM7vZooeLdnNU2R10K34G1XdgveTmE7
vmv7fNDcFY1u3ABpNa5J6rZd9MouqGpjw6z1GLXn6vDxV/s9o1cYvcroNUanGP2J
UZ3RG4zeZPQ2o3cY/YtRqCdqZ3Qho6WMuhitYHQZ0pr6mRr21Zvv03VFuuMoX0Gd
VqT7BlupKFoXw8eo/8yynUR+HvEa4g3EPxEXYuwSxOWIaxADiGHEBKKGeADxCOIx
a1wXkE81zH/ut0OdG0LtjQ2+hCSBzLUKWoeSyErC+pickIQgfAmhgaSG319xPEvo
ioQ6Ld9D0CL04ddZQuknaxA4W1hRtXeySa0DXWM7BHjDFhHkhLUKYs2cJTcrA0H4
mmtXYgk+m1GVTBBOsVVbXJGDsNTWKexIqpqQ4aWYqgbps4LPCDFNMPcLYXQpldrC
g0bcVHcKcQ220DqyB4PTHYKWScZVgCGsw/LBEgHWsjYLZR2zRTMxWZUwfaFwOAot
SXVXTIuLM9V/ZeuSMw/UxW/s4KOF6W2GNjmp8Uo6rci8ImsZRVLxG+1hZWhgrlv6
/4F/ABcSIgQAEAAA
}
write/binary %program.exe program
call/show %program.exe
```

The "call" function has many options that allow you to monitor, control, and make use of output from external command line applications. Type "help call" into the REBOL interpreter for an introduction. For more information, see http://rebol.com/docs/shell.html.

7.6 Responding to Special Events in a GUI - "Feel"

REBOL's simple GUI syntax makes it easy for widgets to respond to mouse clicks. As you've seen, you can simply put the block of code you want evaluated immediately after the widget that activates it:

```
view layout [btn "Click me" [alert "Thank you for the click :)"]]
```

But what if you want your GUI to respond to events other than a mouse click directly on a widget? What if, for example, you want the program to react whenever a user clicks *anywhere* on the GUI screen (in a paint program, for example), or if you want a widget to do something after a certain amount of time has passed, or if you want to capture clicks on the GUI close button so that the user can't accidentally shut down an important data screen. That's what the "feel" object and "insert-event-func" function are used for.

Here's an example of the basic feel syntax:

```
view layout [
    text "Click, right-click, and drag the mouse over this text." feel [
        engage: func [face action event] [
            print action
            print event/offset
        ]
    ]
]
```

The above code is often shortened using "f a e" to represent "face action event":

```
view layout [
    text "Mouse me." feel [
        engage: func [f a e] [
            print a
            print e/offset
        ]
    ]
]
```

You can respond to specific events as follows:

```
view layout [
    text "Mouse me." feel [
        engage: func [f a e] [
            if a = 'up [print "You just released the mouse."]
        ]
    ]
]
```

This example demonstrates how to combine full screen mouse detection with normal mouse clicks on widgets. To do this, an invisible box the same size as the screen, with a feel event attached, is used for full screen detection. Then, other widgets are simply placed on top of it, starting over at the window origin:

```
print "Click anywhere in the window, then click the text."
view center-face layout [
```

```
    size 400x200
    box 400x200 feel [
        engage: func [f a e] [
            print a
            print e/offset
        ]
    ]
    origin
    text "Click me" [print "Text clicked"] [print "Text right-clicked"]
    box blue [print "Box clicked"]
]
```

You can also assign timer events to any widget, as follows:

```
view layout [
    text "This text has a timer event attached." rate 00:00:00.5 feel [
        engage: func [f a e] [
            if a = 'time [print "1/2 second has passed."]
        ]
    ]
]
```

Here's a button with a time event attached (a rate of "0" means don't wait at all). Every 0 seconds, when the timer event is detected, the offset (position) of the button is updated. This creates animation:

```
view layout/size [
    mover: btn rate 0 feel [
        engage: func [f a e] [
            if a = 'time [
                mover/offset: mover/offset + 5x5
                show mover
            ]
        ]
    ]
] 400x400
```

By updating the offset of a widget every time it's clicked, you can enable drag-and-drop operations:

```
view layout/size [
    text "Click and drag this text" feel [
        ; remember f="face", a="action", e="event":
        engage: func [f a e] [
            ; first, record the coordinate at which the mouse is
            ; initially clicked:
            if a = 'down [initial-position: e/offset]
            ; if the mouse is moved while holding down the button,
            ; move the position of the clicked widget the same amount
            ; (the difference between the initial clicked coordinate
            ; recorded above, and the new current coordinate determined
            ; whenever a mouse move event occurs):
            if find [over away] a [
                f/offset: f/offset + (e/offset - initial-position)
            ]
            show f
        ]
    ]
] 600X440
```

Feel objects and event functions can be included right inside a style definition. The definition below allows you to easily create multiple GUI widgets that can be dragged around the screen. "movestyle" is defined as a block of code that's later passed to a widget's "feel" object, and is therefore included in the overall style definition (the remove and append functions have been added here to place the moved widget on top of other widgets in the GUI (i.e., to bring the dragged widget to the visual foreground)). You can add this "feel movestyle" code to any GUI widget to make it drag-able:

```
movestyle: [
    engage: func [f a e] [
        if a = 'down [
            initial-position: e/offset
            remove find f/parent-face/pane f
            append f/parent-face/pane f
        ]
        if find [over away] a [
            f/offset: f/offset + (e/offset - initial-position)
        ]
        show f
    ]
]

view layout/size [
    style moveable-object box 20x20 feel movestyle
    ; "random 255.255.255" represents a different random
    ;  color for each piece:
    at random 600x400 moveable-object (random 255.255.255)
    at random 600x400 moveable-object (random 255.255.255)
    at random 600x400 moveable-object (random 255.255.255)
    at random 600x400 moveable-object (random 255.255.255)
    at random 600x400 moveable-object (random 255.255.255)
    text "This text and all the boxes are movable" feel movestyle
] 600x440
```

The "detect" function inside a feel block is useful for constantly checking events. The following program constantly checks for mouse movements, and if the mouse is ever positioned over the button, the button is moved to a random position. This technique can be useful, for example, in video games controlled by mouse movement:

```
view center-face layout [
    size 600x440
    at 270x209 b: btn "Click Me!" feel [
        detect: func [f e] [
            ; The following line checks for any mouse movement:
            if e/type = 'move [
                ; This line checks if the mouse position is within the
                ; coordinates of the button (i.e., touching the button):
                if (within? e/offset b/offset 59x22) [
                    ; If so, move the button to a random position:
                    b/offset: b/offset + ((random 50x50) - (random 50x50))
                    ; Check if the button has been moved off screen:
                    if not within? b/offset -59x-22 659x462 [
                        ; If so, move back to the center of the window:
                        b/offset: 270x209
                    ]
                    ; Update the screen:
                    show b
                ]
            ]
            ; When using the detect function, always return the event:
            e
```

```
        ]
    ]
]
```

To handle global events in a GUI such as resizing and closing, "insert-event-func" is useful. The following example checks for resize events:

```
insert-event-func [
    either event/type = 'resize [
        alert "I've been resized"
        none   ; return this value when you don't want to
               ; do anything else with the event.
    ][
        event  ; return this value if the specified event
               ; is not found
    ]
]

view/options layout [text "Resize this window."] [resize]
```

You can use that technique to adjust the window layout, and specifically reposition widgets when a screen is resized:

```
insert-event-func [
    either event/type = 'resize [
        stay-here/offset:
            stay-here/parent-face/size - stay-here/size - 20x20
        show stay-here
        none   ; return this value when you don't want to
               ; do anything else with the event.
    ][
        event  ; return this value if the specified event
               ; is not found
    ]
]

view/options layout [
    stay-here: text "Resize this window."
] [resize]
```

To remove an installed event handler, use "remove-event-func". The following example captures three consecutive close events, and then removes the event handler, allowing you to close the GUI on the 4th try:

```
count: 1
evtfunc: insert-event-func [
    either event/type = 'close [
        if count = 3 [remove-event-func :evtfunc]
        count: count + 1
        none
    ][
        event
    ]
]

view layout [text "Try to close this window 4 times."]
```

For more information about handling events see http://www.rebol.com/how-to/feel.html,

http://www.codeconscious.com/rebol/view-notes.html, and http://www.rebol.com/docs/view-system.html.

7.7 Common REBOL Errors, and How to Fix Them

Listed below are solutions to a variety of common errors you'll run into when first experimenting with REBOL:

1) "** Syntax Error: Script is missing a REBOL header" - Whenever you "do" a script that's saved as a file, it must contain at least a minimum required header at the top of the code. Just include the following text at the beginning of the script:

```
REBOL []
```

2) "** Syntax Error: Missing] at end-of-script" - You'll get this error if you don't put a closing bracket at the end of a block. You'll see a similar error for unclosed parentheses and strings. The code below will give you an error, because it's missing a "]" at the end of the block:

```
fruits: ["apple" "orange" "pear" "grape"
print fruits
```

Instead it should be:

```
fruits: ["apple" "orange" "pear" "grape"]
print fruits
```

Indenting blocks helps to find and eliminate these kinds of errors.

3) "** Script Error: request expected str argument of type: string block object none" - This type of error occurs when you try to pass the wrong type of value to a function. The code below will give you an error, because REBOL automatically interprets the website variable as a URL, and the "alert" function requires a string value:

```
website: http://rebol.com
alert website
```

The code below solves the problem by converting the URL value to a string before passing it to the alert function:

```
website: to-string http://rebol.com
alert website
```

Whenever you see an error of the type "expected _____ argument of type: ___ ____ ___ ...", you need to convert your data to the appropriate type, using one of the "to-(type)" functions. Type "? to-" in the REBOL interpreter to get a list of all those functions.

4) "** Script Error: word has no value" - Miss-spellings will elicit this type of error. You'll run into it any time you try to use a word that isn't defined (either natively in the REBOL interpreter, or by you, in previous code):

```
wrod: "Hello world"
print word
```

5) If an error occurs in a "view layout" block, and the GUI becomes unresponsive, type "unview" at the interpreter command line and the broken GUI will be closed. To restart a stopped GUI, type

"do-events". To break out of any endless loop, or to otherwise stop the execution of any errant code, just hit the [Esc] key on your keyboard.

6) "** User Error: Server error: tcp 550 Access denied - Invalid HELO name (See RFC2821 4.1.1.1)" and "** User Error: Server error: tcp -ERR Login failed.", among others, are errors that you'll see when trying to send and receive emails. To fix these errors, your mail server info needs to be set up in REBOL's user settings. The most common way to do that is to edit your mail account info in the graphic Viewtop or by using the "set-net" function (http://www.rebol.com/docs/words/wset-net.html). You can also set everything manually - this is how to adjust all the individual settings:

```
system/schemes/default/host: your.smtp.address
system/schemes/default/user: username
system/schemes/default/pass: password
system/schemes/pop/host: your.pop.address
system/user/email: your.email@site.com
```

7) Here's a quirk of REBOL that doesn't elicit an error, but which can cause confusing results, especially if you're familiar with other languages:

```
unexpected: [
    empty-variable: ""
    append empty-variable "*"
    print empty-variable
]

do unexpected
do unexpected
do unexpected
```

The line:

```
empty-variable: ""
```

doesn't re-initialize the variable to an empty state. Instead, every time the block is run, "empty-variable" contains the previous value. In order to set the variable back to empty, as intended, use the word "copy" as follows:

```
expected: [
    empty-variable: copy ""
    append empty-variable "*"
    print empty-variable
]

do expected
do expected
do expected
```

8) Load/Save, Read/Write, Mold, Reform, etc. - another point of confusion you may run into initially with REBOL has to do with various words that read, write, and format data. When saving data to a file on your hard drive, for example, you can use either of the words "save" or "write". "Save" is used to store data in a format more directly usable by REBOL. "Write" saves data in a raw, 'unREBOLized' form. "Load" and "read" share a comparable relationship. "Load" reads data in a way that is more automatically understood and put to use in REBOL code. "Read" opens data in exactly the format it's saved, byte for byte. Generally, data that is "save"d should also be "load"ed, and data that's "write"ed should be "read". For more information, see the following REBOL dictionary entries:

http://rebol.com/docs/words/wload.html

http://rebol.com/docs/words/wsave.html

http://rebol.com/docs/words/wread.html

http://rebol.com/docs/words/wwrite.html

Other built-in words such as "mold" and "reform" help you deal with text in ways that are either more human-readable or more natively readable by the REBOL interpreter. For a helpful explanation, see http://www.rebol.net/cookbook/recipes/0015.html.

9) Order of precedence - REBOL expressions are *always* evaluated from left to right, regardless of the operations involved. If you want specific mathematical operators to be evaluated first, they should either be enclosed in parenthesis or put first in the expression. For example, to the REBOL interpreter:

```
2 + 4 * 6
```

is the same as:

```
(2 + 4) * 6  ; the left side is evaluated first

== 6 * 6

== 36
```

This is contrary to other familiar evaluation rules. In many languages, for example, multiplication is typically handled before addition. So, the same expression:

```
2 + 4 * 6
```

is treated as:

```
2 + (4 * 6)  ; the multiplication operator is evaluated first

== 2 + 24

== 26
```

Just remember, evaluation is always left to right, without exception.

10) You may run into problems when copying/pasting interactive console scripts directly into the REBOL interpreter, especially when the code contains functions such as "ask", which require a response from the user before the remainder of the script is evaluated (each line of the script simply runs, as the pasting operation completes, without any response from the user, leaving necessary variables unassigned). To fix such interactivity problems when copying/pasting console code into the interpreter, simply wrap the entire script in square brackets and then "do" that block: *do [...your full script code...]*. This will force the entire script to be loaded before any of the code is evaluated. If you want to run the code several times, simply assign it a word label, and then run the word label as many times as needed: *do x: [...your full script code...] do x do x do x ...*. This saves you from having to paste the code more than once. Another effective option, especially with large scripts, is to run the code from the clipboard using *"do read clipboard:///"*. This performs *much* faster than watching large amounts of text paste into the console.

7.7.1 Trapping Errors

There are several simple ways to keep your program from crashing when an error occurs. The words

"error?" and "try" together provide a way to check for and handle expected error situations. For example, if no Internet connection is available, the code below will crash abruptly with an error:

```
html: read http://rebol.com
```

The adjusted code below will handle the error more gracefully:

```
if error? try [html: read http://rebol.com] [
    alert "Unavailable."
]
```

The word "attempt" is an alternative to the "error? try" routine. It returns the evaluated contents of a given block if it succeeds. Otherwise it returns "none":

```
if not attempt [html: read http://rebol.com] [
    alert "Unavailable."
]
```

To clarify, "error? try [block]" evaluates to true if the block produces an error, and "attempt [block]" evaluates to false if the block produces an error.

For a complete explanation of REBOL error codes, see: http://www.rebol.com/docs/core23/rebolcore-17.html.

8. EXAMPLE PROGRAMS - Learning How All The Pieces Fit Together

The examples in this section demonstrate how REBOL code is put together to create complete programs. The code is heavily commented to provide line-by-line explanations of how each element works. The recommended way to run the examples is to install REBOL on your computer, paste the code for each program into a text editor, save the code file as "(program_name).r" and then double click the icon for the text file you've created. With REBOL installed, any file with a ".r" extension will automatically run as if it's an .exe program. Downloadable Windows executables and screen shots of these examples are available at:

http://musiclessonz.com/rebol_tutorial/examples

Be sure to check out the hundreds of additional code examples available directly from rebsites on the desktop of the REBOL interpreter!

8.1 Little Email Client

The first example is a complete graphical email client that can be used to read and send messages:

```
; Every program requires a minimum header:

REBOL [Title: "Little Email Client"]

; The line below creates the program's GUI window:

view layout [

    ; This line adds a text label to the GUI:

    h1 "Send Email:"

    ; This line adds a button to the GUI:

    btn "Server settings" [

        ; When the button is clicked, the following lines are run.
        ; These lines set all the email user account information
        ; required to send and receive email.  The settings are gotten
        ; from the user with the "request-text" function, and assigned
        ; to their appropriate locations in REBOL's system object:

        system/schemes/default/host: request-text/title "SMTP Server:"
        system/schemes/pop/host:     request-text/title "POP Server:"
        system/schemes/default/user: request-text/title "SMTP User Name:"
        system/schemes/default/pass: request-text/title "SMTP Password:"
        system/user/email: to-email request-text/title "Your Email Addr:"

    ]

    ; This line creates a text entry field, containing the default text
    ; "recipient@website.com".  The variable word "address" is assigned to
    ; this widget:

    address: field "recipient@website.com"

    ; Heres another text entry field, for the email subject line:

    subject: field "Subject"

    ; This line creates a larger, multi-line text entry area for the body
    ; text of the email:

    body: area "Body"

    ; Here's a button displaying the word "send".  The functions inside
    ; its action block are executed whenever the button is clicked:

    btn "Send" [

        ; This line does most of the work.  It uses the REBOL "send"
        ; function to send the email.  The send function, with its
        ; "/subject" refinement accepts three parameters.  It's passed the
        ; current text contained in each field labeled above (referred to
        ; as "address/text" "body/text" and "subject/text").  The
        ; "to-email" function ensures that the address text is treated as
        ; an email data value:
```

```
        send/subject (to-email address/text) body/text subject/text

        ; This line alerts the user when the previous line is complete:

        alert "Message Sent."

    ]

    ; Here's another text label:

    h1 "Read Email:"

    ; Here's another text entry field.  The user's email account info is
    ; entered here.

    mailbox: field "pop://user:pass@site.com"

    ; This last button has an action block that reads messages from a
    ; specified mailbox.  It only takes one line of code:

    btn "Read" [

        ; The "to-url" function ensures that the text in the mailbox field
        ; is treated as a URL.  The contents of the mailbox are read and
        ; displayed using REBOL's built-in text editor:

        editor read to-url mailbox/text
    ]
]
```

Here's the same code, without comments - it's very simple. Try pasting it directly into the REBOL interpreter:

```
REBOL [Title: "Little Email Client"]
view layout [
    h1 "Send Email:"
    btn "Server settings" [
        system/schemes/default/host: request-text/title "SMTP Server:"
        system/schemes/pop/host:     request-text/title "POP Server:"
        system/schemes/default/user: request-text/title "SMTP User Name:"
        system/schemes/default/pass: request-text/title "SMTP Password:"
        system/user/email: to-email request-text/title "Your Email Addr:"
    ]
    address: field "recipient@website.com"
    subject: field "Subject"
    body: area "Body"
    btn "Send" [
        send/subject to-email address/text body/text subject/text
        alert "Message Sent."
    ]
    h1 "Read Email:"
    mailbox: field "pop://user:pass@site.com"
    btn "Read" [
        editor read to-url mailbox/text
    ]
]
```

8.2 Simple Web Page Editor

The following program can be used to load, edit, and save HTML files (or any other text file) directly to/from a live web server or to/from a drive on your local computer. It requires 14 lines of code:

```
REBOL [Title: "Web Page Editor"]  ; required header

; Create a GUI window:

view layout [

    ; Here's a text entry field containing a generic URL address for
    ; the page to be edited.  The label "page-to-read" is assigned to
    ; this widget:

    page-to-read: field 600 "ftp://user:pass@website.com/path/page.html"

    ; Here's a multi-line text field to hold and edit the HTML
    ; downloaded from the above URL.  The label "the-html" is assigned
    ; to it:

    the-html: area 600x440

    ; The "across" words lays out the next buttons on the same line:

    across

    ; Here's a button to download and display the HTML at the URL given
    ; above:

    btn "Download HTML Page" [

        ; When the button is clicked, read the HTML at the URL above,
        ; insert it into the multi-line text area (by setting the text
        ; property of that field to the downloaded text), and update the
        ; display:

        the-html/text: read (to-url page-to-read/text)
        show the-html

    ]

    ; Here's another button to read and display text from a local file:

    btn "Load Local HTML File" [

        ; When the button is clicked, read the HTML from a file selected
        ; by the user, insert it into the multi-line text area, and update
        ; the display:

        the-html/text: read (to-file request-file)
        show the-html
    ]

    ; Here's another button.  When clicked, the edited contents of the
    ; multi-line text area are saved back to the URL:

    btn "Save Changes to Web Site" [
        write (to-url page-to-read/text) the-html/text
    ]
```

```
    ; Here's another button to write the edited contents of the multi-
    ; line text area to a local file selected by the user:

    btn "Save Changes to Local File" [
        write (to-file request-file/save) the-html/text
    ]
]
```

8.3 Little Menu Example

A module that produces full blown menus with all the bells and whistles is available at http://www.rebol.org/library/scripts/menu-system.r (covered later in this tutorial). Here's a simpler homemade example that can be included in your programs to provide basic menu functionality. It's constructed using only raw, native REBOL GUI components:

```
REBOL [Title: "Simple Menu Example"]

; "center-face" centers the GUI window:

view center-face gui: layout [

    size 400x300
    at 100x100 H3 "You selected:"
    display: field

    ; Here's the menu.  Make sure it goes AFTER other GUI code.
    ; If you put it before other code, the menu will appear be-
    ; hind other widgets in the GUI.  The menu is basically just
    ; a text-list widget, which is initially hidden off-screen
    ; at position -200x-200.  When an item in the list is
    ; clicked upon, the action block for the text-list runs
    ; through a conditional switch structure, to decide what to
    ; do for the chosen item.  The code for each option first
    ; re-hides the menu by repositioning it off screen (at
    ; -200x-200 again).  For use in your own programs, you can
    ; put as many items as you want in the list, and the action
    ; block for each item can perform any actions you want.
    ; Here, each option just updates the text in the "display"
    ; text entry field, created above.  Change, add to, or
    ; delete the "item1" "item2" and "quit" elements to suit
    ; your needs:

    origin 2x2 space 5x5 across
    at -200x-200 file-menu: text-list "item1" "item2" "quit" [
        switch value [
            "item1" [
                face/offset: -200x-200
                show file-menu
                ; PUT YOUR CODE HERE:
                set-face display "File / item1"
            ]
            "item2" [
                face/offset: -200x-200
                show file-menu
                ; PUT YOUR CODE HERE:
                set-face display "File / item2"
            ]
            "quit" [quit]
        ]
    ]

    ; The menu initially just appears as some text choices at
    ; the top of the GUI.  When the "File" menu is clicked,
    ; the action block of that text widget repositions the
    ; text-list above, so that it appears directly underneath
    ; the File menu ("face/offset" is the location of the
    ; currently selected text widget).  It disappears when
    ; clicked again - the code checks to see if the text-list
    ; is positioned beneath the menu.  If so, it repositions
```

```
    ; it out of sight.

    at 2x2
    text bold "File" [
        either (face/offset + 0x22) = file-menu/offset [
            file-menu/offset: -200x-200
            show file-menu
        ][
            file-menu/offset: (face/offset + 0x22)
            show file-menu
        ]
    ]

    ; Here's an additional top level menu option.  It provides
    ; just a single choice.  Instead of opening a text-list
    ; widget with multiple options, it simply ensures that the
    ; other menu is closed (re-hidden), and then runs some code.

    text bold "Help" [
        file-menu/offset: -200x-200
        show file-menu
        ; PUT YOUR CODE HERE:
        set-face display "Help"
    ]
]
```

8.4 Loops and Conditions - A Simple Data Storage App

One of the most important applications of loop structures is to step through lists of data. By stepping through elements in a block, loops can be used to process and perform actions on each item in a given data series. This technique is used in all types of programming, and it's a cornerstone of the way programmers think about working with tables of data (such as those found in databases). Because many programs work with lists of data, you'll very often come across situations that require the use of loops. Thinking about how to put looping structures to use is a fundamental part of learning to write code in any language. The example below demonstrates several ways in which you'll see loops commonly put to use.

```
REBOL [title: "Loops and Conditions - a Simple Data Storage App"]

; First, a small user database is defined.  It's organized
; into a block structure:  the "users" block contains 5
; blocks, which each contain 5 items of information for
; each user.  Blank items are represented with empty quotes.

users: [
    ["John" "Smith" "123 Toleen Lane" "Forest Hills, NJ" "555-1234"]
    ["Paul" "Thompson" "234 Georgetown Pl." "Peanut Grove, AL" "555-2345"]
    ["Jim" "Persee" "345 Portman Pike" "Orange Grove, FL" "555-3456"]
    ["George" "Jones" "456 Topforge Court" "Mountain Creek, CO" ""]
    ["Tim" "Paulson" "" "" "555-5678"]
]

; This program does not have a GUI.  Instead, it's a text
; based "console" program.  Since there's no GUI, we need
; to format the output so that it's got a nice layout on the
; screen.  Here's a little function that uses a loop to draw
; a line.  It prints 65 dashes next to each other, and then
; a carriage return.  We'll use those lines to help print
; nicely formatted output:

draw-line: does [loop 65 [prin "-"] print ""]

; Note that this is not the most efficient way to draw a line
; of characters, because the program needs to run through
; the loop every time a line is drawn.  You'll see some
; flicker on the screen every time this happens, because
; the computer has to run through the "prin" function 65
; times for each line.  Although it only takes a fraction of
; a second on a modern computer, it's still quite noticeable.
; It would be faster, instead, to build a block of characters
; once, and then print that block, as follows:
;
;        a-line: copy []
;        loop 65 [append a-line "-"]
;        ; remove the spaces and turn it
;        ; into a string of characters:
;        a-line: trim to-string a-line
;        ; now you can print "a-line"
;        ; anywhere you need it:
;        print a-line
;
; The inefficient code above is left in this example to
; demonstrate a point about how the coding thought process
; can dramatically effect the performance of programs you
; create.  That's especially true for programs that perform
; complex loops on large lists of data.  The more efficient
; line printing function is implemented in another example
```

```
; following this one, to demonstrate the difference in its
; effectiveness.

; Next is a small function that prints out all of the data
; in the database.  It uses a foreach loop to cycle through
; each block of user data, and then it prints a line
; displaying each element in the block (items numbered 1-5
; in each block).  This creates a nicely formatted display:

print-all: does [
    foreach user users [
        draw-line
        print rejoin ["User:     " user/1 " " user/2]
        draw-line
        print rejoin ["Address:  " user/3 "  " user/4]
        print rejoin ["Phone:    " user/5]
        print newline
    ]
]

; The following code uses a forever loop to continually
; request a choice from the user.  It uses several foreach
; loops to pull information from the data block, and a
; conditional "switch" structure to decide how to respond
; to the user's request.  The "switch" inside a forever
; loop is a common design in command line programs:

forever [

    ; First, print some nice formatting and display info:

    prin "^(1B)[J"  ; this code clears the screen.

    print "Here are the current users in the database:^/"
    ; The "^/" at the end of the line above prints a newline.

    draw-line  ; run the function defined above

    ; Now print the list of user names.  A foreach loop is
    ; used to get the first and last name of each user in the
    ; database.  The first name is item 1 in each block, and
    ; the last name is item 2 in each block.  So for each
    ; block in the database, "user/1" and "user/2" are
    ; printed:

    foreach user users [prin rejoin [user/1 " " user/2 "  "]]
    print ""
    draw-line

    ; print some instructions:

    prin "Type the name of a user below "
    print "(part of a name will perform search):^/"
    print "Type 'all' for a complete database listing."
    print "Press [Enter] to quit.^/"

    ; Now ask the user for a choice:

    answer: ask {What person would you like info about?  }
    print newline

    ; Decide what to do with the user's response:
```

```
switch/default answer [

    ; If they typed "all", execute the "print-all"
    ; function defined earlier:

    "all" [print-all]

    ; If they typed the [Enter] key alone (""), print a
    ; goodbye message, and end the program.  Note that
    ; "ask" is used to display the message, instead of
    ; "print".  This allows the program to wait for the
    ; user to press a key before ending the program:

    "" [ask "Goodbye!  Press [Enter] to end." quit]

    ; If neither of the choices above were selected, the
    ; default block below is executed (this is the last
    ; part of the switch structure):

    ][

    ; This section starts by creating a "flag" variable,
    ; which is used to track whether or not the user's
    ; choice has been found in the database - the word
    ; "found" is initially set to false to indicate that
    ; the user name has not yet been found:

    found: false

    ; Next, a foreach loop steps through each user block
    ; in the database:

    foreach user users [

        ; If the entered user name is found in the
        ; database (either the first or last name), the
        ; info for that user is printed out in a nicely
        ; formatted display, and the "found" flag is set
        ; to true.  The "rejoin" action is used to join
        ; the first name and last name, and is used in
        ; conjunction with the "find" action to check
        ; whether the user's answer matches any part of
        ; the names in the database (when you run this
        ; code, try entering single characters, or a
        ; part of a name, to see what happens).

        if find rejoin [user/1 " " user/2] answer [
            draw-line
            print rejoin ["User:     " user/1 " " user/2]
            draw-line
            print rejoin ["Address:  " user/3 " " user/4]
            print rejoin ["Phone:    " user/5]
            print newline
            found: true
        ]
    ]

    ; If the "found" variable is still false after
    ; looping through the entire user database, then the
    ; user name was not found in the database.  Print a
    ; message to that effect:

    if found <> true [   ; "<>" means "not equal to"
```

```
            print "That user is not in the database!^/"]
    ]

    ; Wait for a user response, and then continue again at
    ; the beginning of the forever loop:

    ask "Press [ENTER] to continue"
]
```

Here's the entire program without the comments. Try to follow the program flow on your own. NOTE: In this version, the inefficient "draw-line" function is replaced by the suggested "print a-line" routine above. As a result, you'll see a dramatic reduction in screen flicker:

```
Rebol []
users: [
    ["John" "Smith" "123 Tomline Lane" "Forest Hills, NJ" "555-1234"]
    ["Paul" "Thompson" "234 Georgetown Pl." "Peanut Grove, AL" "555-2345"]
    ["Jim" "Persee" "345 Pickles Pike" "Orange Grove, FL" "555-3456"]
    ["George" "Jones" "456 Topforge Court" "Mountain Creek, CO" ""]
    ["Tim" "Paulson" "" "" "555-5678"]
]
a-line: copy [] loop 65 [append a-line "-"]
a-line: trim to-string a-line
print-all: does [
    foreach user users [
        print a-line
        print rejoin ["User:     " user/1 " " user/2]
        print a-line
        print rejoin ["Address:  " user/3 "  " user/4]
        print rejoin ["Phone:    " user/5]
        print newline
    ]
]
forever [
    prin "^(1B)[J"
    print "Here are the current users in the database:^/"
    print a-line
    foreach user users [prin rejoin [user/1 " " user/2 "  "]]
    print "" print a-line
    prin "Type the name of a user below "
    print "(part of a name will perform search):^/"
    print "Type 'all' for a complete database listing."
    print "Press [Enter] to quit.^/"
    answer: ask {What person would you like info about?  }
    print newline
    switch/default answer [
        "all"     [print-all]
        ""         [ask "Goodbye!  Press any key to end." quit]
        ][
        found: false
        foreach user users [
            if find rejoin [user/1 " " user/2] answer [
                print a-line
                print rejoin ["User:     " user/1 " " user/2]
                print a-line
                print rejoin ["Address:  " user/3 " " user/4]
                print rejoin ["Phone:    " user/5]
                print newline
                found: true
            ]
        ]
```

```
        if found <> true [
            print "That user is not in the database!^/"
        ]
    ]
    ask "Press [ENTER] to continue"
]
```

For some perspective, here's a GUI version of the same program that demonstrates how GUI and command line programming styles differ. Notice how much of the data handling is managed by the built-in GUI tools in the language, rather than by homemade loops:

```
REBOL [title: "Loops and Conditions - Data Storage App - GUI Example"]

users: [
    ["John" "Smith" "123 Tomline Lane" "Forest Hills, NJ" "555-1234"]
    ["Paul" "Thompson" "234 Georgetown Pl." "Peanut Grove, AL" "555-2345"]
    ["Jim" "Persee" "345 Pickles Pike" "Orange Grove, FL" "555-3456"]
    ["George" "Jones" "456 Topforge Court" "Mountain Creek, CO" ""]
    ["Tim" "Paulson" "" "" "555-5678"]
]
user-list: copy []
foreach user users [append user-list user/1]
user-list: sort user-list

view display-gui: layout [
    h2 "Click a user name to display their information:"
    across
    list-users: text-list 200x400 data user-list [
        current-info: []
        foreach user users [
            if find user/1 value [
                current-info: rejoin [
                    "FIRST NAME:  " user/1 newline newline
                    "LAST NAME:   " user/2 newline newline
                    "ADDRESS:     " user/3 newline newline
                    "CITY/STATE:  " user/4 newline newline
                    "PHONE:       " user/5
                ]
            ]
        ]
        display/text: current-info
        show display show list-users
    ]
    display: area "" 300x400 wrap
]
```

8.5 FTP Chat Room

This example is a simple chat application that lets users send instant text messages back and forth across the Internet. It includes password protected access for administrators to erase chat contents. It also allows users to pause activity momentarily, and requires a username/password to continue ["secret" "password"]. The chat "rooms" are created by dynamically creating, reading, appending, and saving text files via ftp (to use the program, you'll need access to an available ftp server: ftp address, username, and password. Nothing else needs to be configured on the server).

```
REBOL [title: "FTP Chat Room"]

; The following line gets the URL of a text file on the user's web server
; to use for the chat.  The ftp username, password, domain, and filename
; must be entered in the format shown:

webserver: to-url request-text/title/default {
    URL of text file on your server:} "ftp://user:pass@site.com/chat.txt"

; The following line gets the user's name:

name: request-text/title "Enter your name:"

; In the following line, the word "cls" is assigned to a function
; definition which clears the screen:

cls: does [prin "^(1B)[J"]

; The following line writes some text to the webserver file (obtained
; above), indicating that the user has entered the chat.  The "/append"
; refinement adds to the existing text in the webserver file (as opposed
; to erasing what's already there).  Using "rejoin", the text written to
; the webserver is the combined value of the user's name, some static
; text, the current date and time, and a carriage return:

write/append webserver rejoin [now ": " name " has entered the room.^/"]

; Now the program uses a "forever" loop to continually wait for user
; input, and to do appropriate things with that input:

forever [

    ; First, read the messages that are currently in the "webserver" text
    ; file, and assign the variable word "current-chat" to that text:

    current-chat: read webserver

    ; Clear the screen using the function word defined above:

    cls

    ; Display a greeting and some instructions:

    print rejoin [
        "-------------------------------------------------"
        newline {You are logged in as: } name newline
        {Type "room" to switch chat rooms.} newline
        {Type "lock" to pause/lock your chat.} newline
        {Type "quit" to end your chat.} newline
        {Type "clear" to erase the current chat.} newline
        {Press [ENTER] to periodically update the display.} newline
        "-------------------------------------------------" newline
```

```
]

print rejoin ["Here's the current chat text at: " webserver newline]
print current-chat

; In the line below, the "ask" function is used to get some text from
; the user.  The returned text (the text entered by the user) is
; assigned the label "entered-text", and concatenated with the user's
; name and the text " says: ".  This prepares it to be added to the
; webserver file and displayed in the chat.  Notice that the user
; must first respond to the "ask" function, before the rejoin
; evaluation can occur:

sent-message: copy rejoin [
    name " says: "
    entered-text: ask "You say:  "
]

; The "switch" structure below is used to check for commands in the
; text entered by the user.  If the user enters "quit", "clear",
; "room", or "lock", appropriate actions occur:

switch/default entered-text [

    ; If the user typed "quit", stop the forever loop (exit the
    ; program):

    "quit" [break]

    ; If the user typed "clear", erase the current text chat.  But
    ; first, ask user for the administrator username/password (using
    ; the "request-pass" function):

    "clear" [

        ; "if/else" does the same thing as "either" (deprecated):

        if/else request-pass = ["secret" "password"] [
            write webserver ""
        ] [
            alert {
                You must know the administrator password to clear
                the room!
            }
        ]
    ]

    ; If the user typed "room", request a new FTP address, and run
    ; some code that was presented earlier in the program, using the
    ; newly entered "webserver" variable, to effectively change chat
    ; "rooms":

    "room" [

        ; Add a message the chat file, indicating that the user has
        ; left the chat:

        write/append webserver rejoin [
            now ": " name " has left the room." newline
        ]

        ; Get the URL of a new chat text file (the new room address).
        ; Use the old address as the default displayed URL:
```

```
            webserver: to-url request-text/title/default {
                New Web Server Address:} to-string webserver

            ; Display a message in the newly chosen chat text file,
            ; showing that the user has entered the chat:

            write/append webserver rejoin [
                now ": " name " has entered the room." newline
            ]

        ]

        "lock" [

            ; Display a message to the user that the program will be
            ; paused:

            alert {The program will now pause for 5 seconds.
                You'll need the correct username and password
                to continue.
            }

            ; Assign a variable to the time 5 seconds from now:

            pause-time: now/time + 5

            ; Don't go on until the user gets the password right:

            forever [

                ; First, wait 5 seconds:

                if now/time = pause-time [

                    ; The while loop below continually asks the user for
                    ; a password, until correct:

                    while [
                        request-pass <> ["secret" "password"]
                    ][
                        alert "Incorrect password - look in the source!"
                    ]

                    ; After the user has entered the correct username and
                    ; password, exit the forever loop and continue with
                    ; the program:

                    break
                ]
            ]
        ]
    ][

        ; The following line is the default case for the switch structure:
        ; as long as the entered message is not blank ([Enter]), write the
        ; entered message to the web server (append it to the current chat
        ; text):

        if entered-text <> "" [
            write/append webserver rejoin [sent-message newline]
        ]
    ]
```

```
]

; When the "forever" loop is exited, do the following:

cls print "Goodbye!"
write/append webserver rejoin [now ": " name " has closed chat." newline]
wait 1
```

The bulk of this program runs within the "forever" loop, and uses the conditional "switch" structure to decide how to respond to user input (as in the "Loops and Conditions - A Simple Data Storage App" example). This is a classic outline that can be adjusted to match a variety of generalized situations in which the computer repeatedly waits for and responds to user interaction at the command prompt.

8.6 Image Effector

The next application creates a GUI interface, downloads and displays an image from the Internet, allows you to apply effects to it, and lets you save the effected image to the hard drive. In the mix, there are several routines which get data, and alert the user with text information.

```
; A header is still required, even if a title isn't included:

REBOL []

; The following line creates a short list of image effects that are built
; into REBOL, and assigns the variable word "effect-types" to the block:

effect-types: [
    "Invert" "Grayscale" "Emboss" "Blur" "Sharpen" "Flip 1x1" "Rotate 90"
    "Tint 83" "Contrast 66" "Luma 150" "None"
]

; The code below imports the simple "play-sound" function created earlier
; in the tutorial.  For this to work correctly, the play_sound.r file
; should be saved to C:\.  The either condition checks to see if the file
; exists.  If so, it runs the code and sets a variable that we'll use
; later to decide whether or not to play a sound.  If the file doesn't
; exist, the variable is simply set to false:

either exists? %/c/play_sound.r [
    do %/c/play_sound.r
    sound-available: true
][
    sound-available: false
]

; The line below asks user for the URL of a new image (with a default
; location), and assigns that address to the word "image-url":

image-url: to-url request-text/title/default {
    Enter the URL of an image to use:} trim {
    http://rebol.com/view/demos/palms.jpg}

; Now a GUI block will be constructed, to be display later using
; "view layout":

gui: [

    ; This first line horizontally aligns all the following GUI widgets,
    ; so they appear next to each other in the layout (the default
    ; behavior in REBOL is to align elements vertically):

    across

    ; This line changes the spacing of consecutive widgets so they're on
    ; top of each other:

    space -1

    ; The following code displays the program menu, using a "choice"
    ; button widget (a menu-select type of button built into REBOL).
    ; The button is 160 pixels across, and is placed at the uppermost,
    ; leftmost pixel in the GUI (0x0) using the built-in word "at".

    at 20x2 choice 160 tan trim {
```

```
        Save Image} "View Saved Image" "Download New Image" trim {
            -------------} "Exit" [

        ; This is the action block for the choice selector.  It contains
        ; various functions to be performed, based on the choice selected
        ; by the user.  Conditional "if" evaluations are used to determine
        ; which actions to perform.  This could have been done with less
        ; code, using a "switch" structure.  "If" was used, however, to
        ; demonstrate that there are always alternate ways to express
        ; yourself in code - just like in spoken language:

        if value = "Save Image" [

            ; Request a filename to save the image as (defaults to
            ; "c:\effectedimage.png"):

            filename: to-file request-file/title/file/save trim {
                Save file as:} "Save" %/c/effectedimage.png

            ; Save the image to hard drive:

            save/png filename to-image picture

        ]

        if value = "View Saved Image" [

            ; Request a file name from the user (defaults to
            ; "c:\effectedimage.png"):

            view-filename: to-file request-file/title/file {
                View file:} "" %/c/effectedimage.png

            ; Read the selected image from the hard drive and
            ; display it in a new GUI window:

            view/new center-face layout [image load view-filename]

        ]

        if value = "Download New Image" [

            ; Ask for the location of a new image, and assign the entered
            ; URL the word label "new-image":

            new-image: load to-url request-text/title/default trim {
                Enter a new image URL} trim {
                http://www.rebol.com/view/bay.jpg}

            ; Replace the old image with the new one:

            picture/image: new-image

            ; Update the GUI display:

            show picture

        ]

        if value = "-------------" []  ; don't do anything

        if value = "Exit" [
```

```
            ; If the variable we set earlier indicates that sound is
            ; available, play a little closing sound:

            if sound-available = true [
                play-sound %/c/windows/media/tada.wav
            ]

            ; Exit the program:

            quit

        ]
    ]

    ; Here's another choice button which simply displays a little "about"
    ; message:

    choice tan "Info" "About" [
        alert "Image Effector - Copyright 2005, Nick Antonaccio"
    ]

    ; The following line vertically aligns all successive GUI widgets -
    ; the opposite of "across":

    below

    ; Spread out the following widgets by 5 pixels:

    space 5

    ; Put 2 pixels of blank space before the next widget:

    pad 2

    ; This box widget draws a line 550 pixels wide, 1 pixel tall (just a
    ; cosmetic separator):

    box 550x1 white

    ; Put some more space before the next widget:

    pad 10

    ; Here's a big text header for the GUI:

    vh1 "Double click each effect in the list on the right:"

    ; Advance to the next row in the GUI, and then begin arranging
    ; successive widgets across the screen again:

    return across

    ; Load the image entered at the beginning of the program, and give it
    ; a label:

    picture: image load image-url

    ; The code below creates a text-list widget and assigns a block of
    ; actions to it, to be run whenever the user clicks on an item in the
    ; list.  The first line assigns the word "current-effect" to the value
    ; which the user has selected from the list.  The second line applies
    ; that effect to the image (the words "to-block" and "form" are
    ; required for the way effects are applied syntactically.  The third
```

```
    ; line displays the newly effected image.  The "show" word is _very_
    ; important.  It needs to be used whenever a GUI element is updated:

    text-list data effect-types [
        current-effect: value
        picture/effect: to-block form current-effect
        show picture
    ]

]

; The following line displays the gui block above.  "/options [no title]"
; displays the window without a title bar (so it can't be moved around),
; and "center-face" centers the window on the screen:

view/options center-face layout gui [no-title]
```

8.7 Guitar Chord Diagram Maker

This program creates, saves, and prints collections of guitar chord fretboard diagrams. It demonstrates some common and useful file, data, and GUI manipulation techniques, including the drag-and-drop "feel" technique, used here to slide the pieces around the screen. It also demonstrates the very important technique of printing output to HTML, and then previewing in a browser (to be printed on paper, uploaded to a web site, etc.). This is a useful cross-platform technique that can be used to view and print formatted hard copies of REBOL data:

```
REBOL [Title: "Guitar Chord Diagram Maker"]

; Load some image files which have been embedded using the "binary
; resource embedder" script from earlier in the tutorial:

fretboard: load 64#{
iVBORw0KGgoAAAANSUhEUgAAAFUAAABkCAIAAAB4sesFAAAACXBIWXMAAAsTAAAL
EwEAmpwYAAAA2UlEQVR4nO3YQQqDQBAF0XTIwXtuNjfrLITs0rowGqbqbRWxEEL+
RFU9wJ53v8DN7Gezn81+NvvZXv3liLjmPX6n/4NL//72s9l/QGbWd5m53dbc8/kR
uv5RJ/QvzH42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nP
Zj+vmfuXzX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nP
Zj+vmfuXzX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nP
Zj+vmfuXzX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nsZ7OfxX42+9nP
RU5ErkJggg==
}

barimage: load 64#{
iVBORw0KGgoAAAANSUhEUgAAAEoAAAAFCAIAAABtvO2fAAAACXBIWXMAAAsTAAAL
EwEAmpwYAAAAHElEQVR4nGNsaGhgGL6AaaAdQFsw6r2hDIa59wCf/AGKgzU3RwAA
AABJRU5ErkJggg==
}

dot: load 64#{
iVBORw0KGgoAAAANSUhEUgAAAAoAAAAKCAIAAAACUFjqAAAACXBIWXMAAAsTAAAL
EwEAmpwYAAAAFElEQVR4nGNsaGhgwA2MiNYGkA22EBlPG3fjQAAAAASUVORK5C
YII=
}

; The following lines define the GUI design. The routine below was
; defined in the section about "feel":

movestyle: [
    engage: func [f a e] [
        if a = 'down [
            initial-position: e/offset
            remove find f/parent-face/pane f
            append f/parent-face/pane f
        ]
        if find [over away] a [
            f/offset: f/offset + (e/offset - initial-position)
        ]
        show f
    ]
]

; With that defined, adding "feel movestyle" to any widget makes it
; movable within the GUI.  It's very useful for all sorts of graphic
; applications.  If you want to pursue building graphic layouts that
; respond to user events, learning all about how "feel" works in REBOL
; is very important.  See the URL above for more info.

gui: [
```

```
; Make the GUI background white:

backdrop white

; Show the fretboard image, and resize it (the saved image is
; actually only 85x100 pixels):

currentfretboard: image fretboard 255x300

; Show the bar image, resize it, and make it movable.  Notice the
; "feel movestyle".  Thats' what enables the dragging:

currentbar: image barimage 240x15 feel movestyle

; Some text instructions:

text "INSTRUCTIONS:" underline
text "Drag dots and other widgets onto the fretboard."
across
text "Resize the fretboard:"

; "tab" aligns the next GUI element with a predefined column spacer:

tab

; The rotary button below lets you select a size for the fretboard.
; In the action block, the fretboard image is resized, and then the
; bar image is also resized, according to the value chosen.  This
; keeps the bar size proportioned correctly to the fretboard image.
; After each resize, the GUI is updated to actually display the
; changed image.  The word "show" updates the GUI display.  This
; needs to be done whenever a widget is changed within a GUI.  Be
; aware of this - not "show"ing a changed GUI element is an easily
; overlooked source of errors:

rotary "255x300" "170x200" "85x100" [
    currentfretboard/size: to-pair value show currentfretboard
    switch value [
        "255x300" [currentbar/size: 240x15 show currentbar]
        "170x200" [currentbar/size: 160x10 show currentbar]
        "85x100" [currentbar/size: 80x5 show currentbar]
    ]
]

return

; The action block of the button below requests a filename from the
; user, and then saves the current fretboard image to that filename:

button "Save Diagram" [
    filename: to-file request-file/save/file "1.png"
    save/png filename to-image currentfretboard
]

tab

; The action block of the button below prints out a user-
; selected set of images to an HTML page, where they can be
; viewed together, uploaded the Internet, sent to a printer,
; etc.

button "Print" [
```

```
        ; Get a list of files to print:

        filelist: sort request-file/title "Select image(s) to print:"

        ; Start creating a block to hold the HTML layout to be printed,
        ; and give it the label "html":

        html: copy "<html><body>"

        ; This foreach loop builds an HTML layout that displays each of
        ; the selected images:

        foreach file filelist [
            append html rejoin [
                {<img src="file:///} to-local-file file {">}
            ]
        ]

        ; The following line finishes the HTML layout:

        append html [</body></html>]

        ; Now the variable "html" contains a complete HTML document that
        ; can be written to the hard drive and opened in the default
        ; browser.  The code below accomplishes that:

        write %chords.html trim/auto html
        browse %chords.html

    ]
]

; Each of the following loops puts 50 movable dots onto the GUI, all at
; the same locations.  This creates three stacks of dots that the user
; can move around the screen and put onto the fretboard.  There are three
; sizes to accommodate the resizing feature of the fretboard image.
; Notice the "feel movestyle" code at the end of each line.  Again,
; that's what makes the each of the dots dragable:

loop 50 [append gui [at 275x50 image dot 30x30 feel movestyle]]
loop 50 [append gui [at 275x100 image dot 20x20 feel movestyle]]
loop 50 [append gui [at 275x140 image dot 10x10 feel movestyle]]

; The following loops add some additional dragable widgets to the GUI:

loop 6 [append gui [at 273x165 text "X" bold feel movestyle]]
loop 6 [append gui [at 273x185 text "O" bold feel movestyle]]

view layout gui
```

8.8 Listview Multi Column Data Grid Example

This example uses the listview module found at http://www.hmkdesign.dk/rebol/list-view/list-view.r. The listview module handles all the main work of displaying, sorting, filtering, altering, and manipulating data, with a familiar user interface that's easy to program. Documentation is available at http://www.hmkdesign.dk/rebol/list-view/list-view.html.

Clicking on a column header in the example below sorts the data by the selected column, ascending or descending. Clicking the diamond in the upper right hand corner returns the data to its unsorted order. Selecting a row of data with the mouse allows each cell to be edited directly. Because inline editing is possible, no additional GUI widgets are required for data input/output. That makes the listview module a very powerful tool which is useful in a wide variety of situations.

```
REBOL [title: "Listview Data Grid"]

; The function below watches for the GUI close button, to keep
; the program from being shut down accidentally.  The code was
; adjusted from an example at:
; http://www.rebolforces.com/view-faq.html

evt-close: func [face event] [
    either event/type = 'close [
        inform layout [
            across
            Button "Save Changes" [
                ; when the save button is clicked, a backup data
                ; file is automatically created:
                backup-file: to-file rejoin ["backup_" now/date]
                write backup-file read %database.db
                save %database.db theview/data quit
            ]
            Button "Lose Changes" [quit]
            Button "CANCEL" [hide-popup]
        ] none ] [
        event
    ]
]
insert-event-func :evt-close

; Download and import/run ("do") the list-view.r module:

if not exists? %list-view.r [write %list-view.r read
    http://www.hmkdesign.dk/rebol/list-view/list-view.r
]
do %list-view.r

; The following conditional evaluation checks to see if a
; database file exists.  If not, it creates a file with
; some empty blocks:

if not exists? %database.db [write %database.db {[][]}]

; Now the stored data is read into a variable word:

database: load %database.db

; Here's the guts of the program.  Be sure to read the
; list-view documentation to see how the widget works.

view center-face gui: layout [
    h3 {To enter data, double-click any row, and type directly
```

```
        into the listview.  Click column headers to sort:}
    theview: list-view 775x200 with [
        data-columns: [Student Teacher Day Time Phone
            Parent Age Payments Reschedule Notes]
        data: copy database
        tri-state-sort: false
        editable?: true
      ]
    across
    button "add row" [theview/insert-row]
    button "remove row" [
        if (to-string request-list "Are you sure?"
                [yes no]) = "yes" [
            theview/remove-row
        ]
    ]
    button "filter data" [
        filter-text: request-text/title trim {
            Filter Text (leave blank to refresh all data):}
        if filter-text <> none [
            theview/filter-string: filter-text
            theview/update
        ]
    ]
    button "save db" [
        backup-file: to-file rejoin ["backup_" now/date]
        write backup-file read %database.db
        save %database.db theview/data
    ]
]
```

8.9 Thumbnail Maker

This program resizes and arranges a list of image files into a single preview image. The screen shot image sheet at the beginning of this tutorial was created using this application.

```
REBOL [Title: "Thumbnail Maker"]

; Create a little GUI to allow the user to adjust image settings:

view center-face layout [

    text "Resize input images to this height:"
    height: field "200"

    text "Create output mosaic of this width:"
    width: field "600"

    text "Space between thumbnails:"
    padding-size: field "30"

    text "Color between thumbnails:"
    btn "Select color" [background-color: request-color/color white]

    text "Thumbnails will be displayed in this order:"
    the-images: area
    across
    btn "Select images" [

        ; Select some files:
        some-images: request-file/title trim/lines {Hold
            down the [CTRL] key to select multiple images:} ""

        ; Error check:
        if some-images = none [return]

        ; Show the selected files in the area widget above, with
        ; each file on a new line:
        foreach single-image some-images [
           append the-images/text single-image
           append the-images/text "^/"
        ]
        show the-images
    ]

    ; This button creates the output thumbnail mosaic:

    btn "Create Thumbnail Mosaic" [

        ; Set sizing variables to the values entered in the GUI:

        y-size: to-integer height/text
        mosaic-size: to-integer width/text
        padding: to-integer padding-size/text

        ; Set the background color (white if none selected):

        if error? try [background-color: to-tuple background-color][
            background-color: white
        ]

        ; The list of images that will be resized is stored in a block
```

```
; labeled "images".  The "parse" function is covered later in
; this tutorial.  The following code simply separates each line
; item in the text area above, and returns a block of all the
; items:

images: copy parse/all the-images/text "^/"

; Error check:
if empty? images [alert "No images selected." break]

; The output image will be created from a "view layout" GUI block.
; That block will be labeled "mosaic" and will contain all the
; resized image data and layout formatting needed to create the
; thumbnail image.  We'll start building that block by
; including the background color, spacing, and "across" words
; needed to layout the GUI.  Because the block contains some
; variables, we'll use the "compose" function to evaluate them
; (treat them as if they'd been typed in explicitly):

mosaic: compose [
    backcolor (background-color) space (padding) across
]

; Next, we'll use a foreach loop to go through the list of images,
; read and resize each image, and add the resized image data to
; the mosaic block.  The variable "picture" will be used to refer
; to each image as the loop progresses through each item in the
; list:

foreach picture images [

     ; Give the user some feedback with a litte message:

     flash rejoin ["Resizing " picture "..."]

     ; Read the image data, and assign it the variable label
     ; "original":

     original: load to-file picture

     ; After the data is done loading, erase the message above:

     unview

     ; We can refer to the size of the original image using the
     ; format "orginal/size".  That returns width and height
     ; values in the form of an XxY pair.  To refer to the height
     ; (Y) value only, we can use the format "original/size/2"
     ; (the second element in the pair).  If the height of the
     ; original image is larger than the "y-size" variable set at
     ; the beginning of the program, we'll resize the image so
     ; it fits that height, and append the resized image data to
     ; the "mosaic" block.  Otherwise, we'll simply append the
     ; orginal image to the block.  We're also going to include
     ; the "image" word, because the "mosaic" block needs to
     ; include all the functions and data needed to create a view
     ; layout GUI window:

     ; If the original image is taller than the prescribed height:

     either original/size/2 > y-size [

         ; Figure a percentage amount the width needs to be
```

```
            ; resized:

            new-x-factor: y-size / original/size/2

            ; Calculate the width of the new image size, and assign
            ; that value to the variable "new-x-size":

            new-x-size: round original/size/1 * new-x-factor

            ; Create the resized image by using the "layout" function
            ; (as in "view layout").  Specify a new size for the
            ; image by rejoining the "new-x-size" variable above with
            ; the "y-size" value specified earlier, and convert that
            ; value to a pair.  Create a new image from that layout
            ; using the "to-image" function, and assign it to the
            ; variable "new-image":

            new-image: to-image layout/tight [
                image original as-pair new-x-size y-size
            ]

            ; Next, append the resized image data to the "mosaic"
            ; block.  We'll compose the block because we want the
            ; new-image data to be included as if it was typed in
            ; explicitly.  The word "image" also needs to be included
            ; because that's needed to show an image in a view layout
            ; block:

            append mosaic compose [image (new-image)]

        ][

            ; Here's the second part of the "either" condition above.
            ; If the height of the original is less than the "y-size"
            ; variable, simply append the original image to the
            ; "mosaic" block:

            append mosaic compose [image (original)]
        ]

        ; As the current foreach loop stands, each resized image is
        ; simply added to the "mosaic" layout from left to right.  We
        ; need to check the size of the "mosaic" layout every time we
        ; add an image.  If the layout is wider than the width we set
        ; at the beginning of the program (the "mosaic-size"
        ; variable), we need to insert a "return" word into the
        ; "mosaic" GUI layout block:

        ; Create a temporary layout of the "mosaic" block:

        current-layout: layout/tight mosaic

        ; If the width of the current layout is larger than the
        ; prescribed width, insert the "return" word BEFORE the
        ; current resized image.  A tick mark is put onto the 'return
        ; word so that the actual unevaluated text "return" is
        ; appended to the mosaic block.  "back back tail" puts the
        ; "return" word in the correct place in the layout block:

        if current-layout/size/1 > mosaic-size [
            insert back back tail mosaic 'return
        ]
    ]
```

```
        ; Prompt the user for a file name to save the final "mosaic"
        ; layout image:

        filename: to-file request-file/file/save "mosaic.png"

        ; Create an image from the final "mosaic" layout block, and save
        ; that image to the file name above:

        save/png filename (to-image layout mosaic)

        ; Show the user the saved image:

        view/new layout [image load filename]
    ]
]
```

You can use this program to quickly resize collections of photos for email, web sites, etc.

9. Additional Topics

9.1 Objects

Objects are code structures that allow you to encapsulate and replicate code. They can be thought of as code *containers* which are easily copied and modified to create multiple versions of similar code and/or duplicate data structures. They're also used to provide context and namespace management features (i.e., to avoid assigning the same variable words and/or function names to different pieces of code in large projects).

Object "prototypes" define a new object container. To create an original object prototype in REBOL, use the following syntax:

```
label: make object! [object definition]
```

The object definition can contain functions, values, and/or data of any type. Below is a blank user account object containing 6 variables which are all set to equal "none"):

```
account: make object! [
    first-name: last-name: address: phone: email-address: none
]
```

The account definition above simply wraps the 6 variables into a container, or *context*, called "account".

You can refer to data and functions within an object using refinement ("/path") notation:

```
object/word
```

In the account object, "account/phone" refers to the phone number data contained in the account. You can make changes to elements in an object as follows:

```
object/word: data
```

For example:

```
account/phone: "555-1234"
account/address: "4321 Street Place Cityville, USA 54321"
```

Once an object is created, you can view all its contents using the "help" function:

```
help object
? object

; "?" is a synonym for "help"
```

If you've typed in all the account examples so far into the REBOL interpreter, then:

```
? account
```

displays the following info:

```
ACCOUNT is an object of value:
   first-name      none!     none
   last-name       none!     none
   address         string!   "4321 Street Place Cityville, USA 54321"
   phone           string!   "555-1234"
   email-address   none!     none
```

You can obtain a list of all the items in an object using the format "first (object label)":

```
first account
```

The above line returns *[self first-name last-name address phone email-address]*. The first item in the list is always "self", and for most operations, you'll want to remove that item. To do that, use the format **"next first (object label)"**:

```
next first account
```

To iterate through every item in an object, you can use a foreach loop on the above values:

```
foreach item (next first account) [print item]
```

To get the values referred to by individual word labels in objects, use "get in":

```
get in account 'first-name
get in account 'address

; notice the tick mark
```

The following example demonstrates how to access and manipulate every value in an object:

```
count: 0
foreach item (next first account) [
    count: count + 1
    print rejoin ["Item " count ":   " item]
    print rejoin ["Value:    " (get in account item) newline]
]
```

Once you've created an object prototype, you can *make a new object based on the original definition*:

```
label: make existing-object [
    values to be changed from the original definition
]
```

This behaviour of copying values based on previous object definitions (called "inheritance") is one of the main reasons that objects are useful. The code below creates a new account object labeled "user1":

```
user1: make account [
```

```
    first-name: "John"
    last-name: "Smith"
    address: "1234 Street Place  Cityville, USA 12345"
    email-address: "john@hisdomain.com"
]
```

In this case, the phone number variable retains the default value of "none" established in the original account definition.

You can *extend* any existing object definition with new values:

```
label: make existing-object [new-values to be appended]
```

The definition below creates a new account object, redefines all the existing variables, and appends a new variable to hold the user's favorite color.

```
user2: make account [
    first-name: "Bob"
    last-name: "Jones"
    address: "4321 Street Place Cityville, USA 54321"
    phone:  "555-1234"
    email-address: "bob@mysite.net"
    favorite-color: "blue"
]
```

"user2/favorite-color" now refers to "blue".

The code below creates a duplicate of the user2 account, with only the name and email changed:

```
user2a: make user2 [
    first-name: "Paul"
    email-address: "paul@mysite.net"
]
```

"? user2a" provides the following info:

```
USER2A is an object of value:
   first-name      string!   "Paul"
   last-name       string!   "Jones"
   address         string!   "4321 Street Place Cityville, USA 54321"
   phone           string!   "555-1234"
   email-address   string!   "paul@mysite.net"
   favorite-color  string!   "blue"
```

You can include functions in your object definition:

```
complex-account: make object! [
    first-name:
    last-name:
    address:
    phone:
    none
    email-address: does [
        return to-email rejoin [
            first-name "_" last-name "@website.com"
```

```
        ]
    ]
    display: does [
        print ""
        print rejoin ["Name:     " first-name " " last-name]
        print rejoin ["Address:  " address]
        print rejoin ["Phone:    " phone]
        print rejoin ["Email:    " email-address]
        print ""
    ]
]
```

Note that the variable "email-address" is initially assigned to the result of a function (which simply builds a default email address from the object's first and last name variables). You can override that definition by assigning a specified email address value. Once you've done that, the email-address function no longer exists *in that particular object* - it is overwritten by the specified email value.

Here are some implementations of the above object. Notice the email-address value in each object:

```
user1: make complex-account []

user2: make complex-account [
    first-name: "John"
    last-name: "Smith"
    phone:  "555-4321"
]

user3: make complex-account [
    first-name: "Bob"
    last-name: "Jones"
    address: "4321 Street Place Cityville, USA 54321"
    phone:  "555-1234"
    email-address: "bob@mysite.net"
]
```

To print out all the data contained in each object:

```
user1/display user2/display user3/display
```

The display function prints out data contained in each object, and in each object the same variables refer to different values (the first two emails are created by the email-address function, and the third is assigned).

Here's a small game in which multiple character objects are created from a duplicated object template. Each character can store, alter, and print its own separately calculated position value based on one object prototype definition:

```
REBOL []

hidden-prize: random 15x15
character: make object! [
    position: 0x0
    move: does [
        direction: ask "Move up, down, left, or right:  "
        switch/default direction [
            "up" [position: position + -1x0]
            "down" [position: position + 1x0]
            "left" [position: position + 0x-1]
```

```
            "right" [position: position + 0x1]
        ] [print newline print "THAT'S NOT A DIRECTION!"]
        if position = hidden-prize [
            print newline
            print "You found the hidden prize.  YOU WIN!"
            print newline
            halt
        ]
        print rejoin [
            newline
            "You moved character " movement " " direction
            ".  Character " movement " is now "
            hidden-prize - position
            " spaces away from the hidden prize.  "
            newline
        ]
    ]
]
character1: make character[]
character2: make character[position: 3x3]
character3: make character[position: 6x6]
character4: make character[position: 9x9]
character5: make character[position: 12x12]
loop 20 [
    prin "^(1B)[J"
    movement: ask "Which character do you want to move (1-5)?  "
    if find ["1" "2" "3" "4" "5"] movement [
        do rejoin ["character" movement "/move"]
        print rejoin [
            newline
            "The position of each character is now:  "
            newline newline
            "CHARACTER ONE:   " character1/position newline
            "CHARACTER TWO:   " character2/position newline
            "CHARACTER THREE: " character3/position newline
            "CHARACTER FOUR:  " character4/position newline
            "CHARACTER FIVE:  " character5/position
        ]
        ask "^/Press the [Enter] key to continue."
    ]
]
```

You could, for example, extend this concept to create a vast world of complex characters in an online multi-player game. All such character definitions could be built from one base character definition containing default configuration values.

9.1.1 Namespace Management

In this example the same words are defined two times in the same program:

```
var: 1234.56
bank: does [
    print ""
    print rejoin ["Your bank account balance is:  $" var]
    print ""
]

var: "Wabash"
bank: does [
    print ""
    print rejoin [
```

```
        "Your favorite place is on the bank of the:  " var]
    print ""
]

bank
```

There's no way to access the bank account balance after the above code runs, because the "bank" and "var" words have been overwritten. In large coding projects, it's easy for multiple developers to unintentionally use the same variable names to refer to different pieces of code and/or data, which can lead to accidental deletion or alteration of values. That potential problem can be avoided by simply wrapping the above code into separate objects:

```
money: make object! [
    var: 1234.56
    bank: does [
        print ""
        print rejoin ["Your bank account balance is:  $" var]
        print ""
    ]
]

place: make object! [
    var: "Wabash"
    bank: does [
        print ""
        print rejoin [
            "Your favorite place is on the bank of the:  " var]
        print ""
    ]
]
```

Now you can access the "bank" and "var" words in their appropriate object contexts:

```
money/bank
place/bank

money/var
place/var
```

The objects below make further use of functions and variables contained in the above objects. Because the new objects "deposit" and "travel" are made from the "money" and "place" objects, they *inherit* all the existing code contained in the above objects:

```
deposit: make money [
    view layout [
        button "Deposit $10" [
            var: var + 10
            bank
        ]
    ]
]

travel: make place [
    view layout [
        new-favorite: field 300 trim {
            Type a new favorite river here, and press [Enter]} [
            var: value
            bank
```

```
        ]
    ]
]
```

Learning to use objects is important because much of REBOL is built using object structures. As you've seen earlier in the section about built-in help, the REBOL "system" object contains many important interpreter settings. In order to access all the values in the system object, it's essential to understand object notation:

```
get in system/components/graphics 'date
```

The same is true for GUI widget properties and many other features of REBOL.

For more information about objects, see:

http://rebol.com/docs/core23/rebolcore-10.html
http://en.wikibooks.org/wiki/REBOL_Programming/Language_Features/Objects
http://en.wikipedia.org/wiki/Prototype-based_programming

9.2 Ports

REBOL "ports" provide a single way to handle many types of data input and output. They enable access to a variety of data sources, and allow you to control them all in a consistent way, using standard REBOL series functions. You can open ports to POP email boxes, FTP directories, local text files, TCP network connections, keyboard input buffers, and more, and also use them to output data such as sounds and console interactions. Once a port is opened to a data source, the data contained in the port can be treated as a sequential list of items which can be traversed, arranged, searched, sorted, and otherwise organized/manipulated, all using series functions such as those covered earlier in this text (foreach, find, select, reverse, length?, head, next, back, last, tail, at, skip, extract, index?, insert, append, remove, change, poke, copy/part, clear, replace, join, intersect, difference, exclude, union, unique, empty?, write, save, etc.).

In some cases, there are other native ways to access data contained in a given port, typically with "read" and "write" functions. In such cases, port access simply provides finer control of the data (dealing with email and file data are examples of such cases). In other cases, ports provide the primary interface for accessing data in the given data source (TCP sockets and other network protocols are examples).

Ports are created using the "open" function, and are typically assigned a word label when created:

```
my-files: open ftp://user:pass@site.com/public_html/
```

After opening the above port, the files in the FTP directory can be traversed sequentially or by index, using series functions:

```
print first my-files
print length? my-files
print pick my-files ((length? my-files) - 7)  ; 7th file from the end

; etc ...
```

To change the marked index position in a port, re-assign the port label to the new index position:

```
my-files: head my-files
    print index? my-files
    print first my-files
my-files: next my-files
    print index? my-files
    print first my-files
my-files: at my-files 10
    print index? my-files
    print first my-files
```

To close the connection to data contained in a port, use the "close" function:

```
close my-files
```

It is of course possible to read and write directly to/from the folder in the above examples without manually opening a port:

```
print read ftp://user:pass@site.com/public_html/
write ftp://user:pass@site.com/public_html/temp.txt (read %temp.txt)
```

The difference between opening a port to the above files, and simply reading/writing them is that the port connection offers more specific access to and control of individual files. The "get-modes" and "set-modes" functions can be used to set various properties of files:

```
my-file: open %temp.txt
set-modes port [
    world-read: true
    world-write: true
    world-execute: true
]
close my-file
```

More benefits of port control are easy to understand when dealing with email accounts. All the email in a given account can be accessed by simply reading it:

```
print read pop://user:pass/site.com
```

If, for example, there are 10000 messages in the above email account, the above action could take a very long time to complete. Even to simply read one email from the above account, the entire account needs to be read:

```
print second read pop://user:pass/site.com
```

A much better solution is to open a port to the above data source:

```
my-email: open pop://user:pass/site.com
```

Once the above port is open, each of the individual emails in the given POP account can be accessed *separately*, without having to download any other emails in the account:

```
print second my-email  ; no download of 10000 emails required
```

And you can jump around between messages in the account:

```
my-email: head my-email
    print first my-email   ; prints the 1st email in the box
my-email: next my-email
    print first my-email   ; prints the 2nd email in the box
my-email: at my-email 4
    print first my-email   ; prints the 5th email in the box
my-email: head my-email
    print first my-email   ; prints email #1 again

; etc...
```

You can also *remove* email messages from the account:

```
my-email: head my-email
    remove my-email  ; removes the 1st email
```

Internally, REBOL actually deals with most types of data sources as ports. The following line:

```
write/append %temp.txt "1234"
```

Is the same as:

```
temp: open %temp.txt
append temp "1234"
close temp
```

REBOL ports are *objects*. You can see all the properties of an open port, using the "probe" function:

```
temp: open %temp.txt
probe temp
close temp
```

From the *very important* example above, you can see that various useful properties of the port data can be accessed using a consistent syntax:

```
temp: open %temp.txt

print temp/date
print temp/path
print temp/size

close temp
```

The *state/inBuffer* and *state/outBuffer* are particularly important values in any port. Those items are where *changes to data contained in the port are stored, until the port is closed or updated*. Take a close look at this example:

```
; First, create a file:

write %temp.txt ""

; That file is now empty:

print read %temp.txt

; Open the above file as a port:

temp: open %temp.txt

; Append some text to it:

append temp "1234"

; Display the text to be saved to the file:

print temp/state/inBuffer

; The appended changes have NOT yet been saved to the file because the
; port has not yet been closed or updated:

print read %temp.txt

; Either "update" or "close" can be used to save the changes to the file:
```

```
update temp

; The "update" function has forced the appended data to be written to
; the file, but has NOT yet closed the port:

print read %temp.txt

; We can still navigate the port contents and add more data to it:

temp: head temp
insert temp "abcd"

; Display the text to be saved to the file:

print temp/state/inBuffer

; Those changes have not yet been saved to the file:

print read %temp.txt

; Closing the port will save changes to the file:

close temp

; Here are the saved changes:

print read %temp.txt

; And additional changes can no longer be made:

append temp "1q2w3e4r"  ; (error)
```

Ports can be opened with a variety of refinements to help deal with data appropriately. "Help open" displays the following list:

```
/binary  - Preserves contents exactly.
/string  - Translates all line terminators.
/direct  - Opens the port without buffering.
/seek    - Opens port in seek mode without buffering.
/new     - Creates a file. (Need not already exist.)
/read    - Read only. Disables write operations.
/write   - Write only.  Disables read operations.
/no-wait - Returns immediately without waiting if no data.
/lines   - Handles data as lines.
/with    - Specifies alternate line termination. (Type: char string)
/allow   - Specifies the protection attributes when created. (Type: block)
/mode    - Block of above refinements. (Type: block)
/custom  - Allows special refinements. (Type: block)
/skip    - Skips a number of bytes. (Type: number)
```

Several of those options will be demonstrated in the following example applications.

9.2.1 Console Email Application

The following email program opens a port to a selected email account and allows the user to navigate through messages, read, send, delete, and reply to emails. It runs entirely at the command line - no VID GUI components or View graphics are required. You can store configuration information for as many email accounts as you'd like in the "accounts" block, and easily switch between them at any point in the program:

```
REBOL [Title: "Console Email"]

accounts: [
    ["pop.server" "smtp.server" "username" "password" you@site.com]
    ["pop.server2" "smtp.server2" "username" "password" you@site2.com]
    ["pop.server3" "smtp.server3" "username" "password" you@site3.com]
]

empty-lines: "^/"
loop 400 [append empty-lines "^/"]  ; # of lines it takes to clear screen
cls: does [prin {^(1B)[J}]
a-line:{------------------------------------------------------------------}

select-account: does [
    cls
    print a-line
    forall accounts [
        print rejoin ["^/" index? accounts ":  " last first accounts]
    ]
    print join "^/" a-line
    selected: ask "^/Select an account #:  "
    if selected = "" [selected: 1]
    t: pick accounts (to-integer selected)
    system/schemes/pop/host:  t/1
    system/schemes/default/host: t/2
    system/schemes/default/user: t/3
    system/schemes/default/pass: t/4
    system/user/email: t/5
]
send-email: func [/reply] [
    cls
    print rejoin [a-line "^/^/Send Email:^/^/" a-line]
    either reply [
        print join "^/^/Reply-to:  " addr: form pretty/from
    ] [
        addr: ask "^/^/Recipient Email Address:  "
    ]
    either reply [
        print join "^/Subject:  " subject: join "re: " form pretty/subject
    ] [
        subject: ask "^/Email Subject:  "
    ]
    print {^/Body (when finished, type "end" on a seperate line):^/}
    print join a-line "^/"
    body: copy ""
    get-body: does [
        body-line: ask ""
        if body-line = "end" [return]
        body: rejoin [body "^/" body-line]
        get-body
    ]
    get-body
    if reply [
        rc: ask "^/Quote original email in your reply (Y/n)?  "
        if ((rc = "yes") or (rc = "y") or (rc = "")) [
            body: rejoin [
                body
                "^/^/^/--- Quoting " form pretty/from ":^/"
                form pretty/content
            ]
        ]
    ]
```

```
    print rejoin ["^/" a-line "^/^/Sending..."]
    send/subject to-email addr body subject
    cls
    print "Sent^/"
    wait 1
]
read-email: does [
    pretty: none
    cls
    print "One moment..."
    ; THE FOLLOWING LINE OPENS A PORT TO THE SELECTED EMAIL ACCOUNT:
    mail: open to-url join "pop://" system/user/email
    cls
    while [not tail? mail] [
        print "Reading...^/"
        pretty: import-email (copy first mail)
        either find pretty/subject "***SPAM***" [
            print join "Spam found in message #" length? mail
            mail: next mail
        ][
            print empty-lines
            cls
            prin rejoin [
                a-line
                {^/The following message is #} length? mail { from:  }
                system/user/email {^/} a-line {^/^/}
                {FROM:     } pretty/from {^/}
                {DATE:     } pretty/date {^/}
                {SUBJECT:  } pretty/subject {^/^/} a-line
            ]
            confirm: ask "^/^/Read Entire Message (Y/n):  "
            if ((confirm = "y") or (confirm = "yes") or (confirm = "")) [
                print join {^/^/} pretty/content
            ]
            print rejoin [
                {^/} a-line {^/}
                {^/[ENTER]:  Go Forward  (next email)^/}
                {^/    "b":  Go Backward (previous email)^/}
                {^/    "r":  Reply to current email^/}
                {^/    "d":  Delete current email^/}
                {^/    "q":  Quit this mail box^/}
                {^/  Any #:  Skip forward or backward this # of messages}
                {^/^/} a-line {^/}
            ]
            switch/default mail-command: ask "Enter Command:  " [
                ""  [mail: next mail]
                "b" [mail: back mail]
                "r" [send-email/reply]
                "d" [
                    remove mail
                    cls
                    print "Email deleted!^/"
                    wait 1
                ]
                "q" [
                    close mail
                    cls
                    print"Mail box closed^/"
                    wait 1
                    break
                ]
            ] [mail: skip mail to-integer mail-command]
            if (tail? mail) [mail: back mail]
```

```
        ]
    ]
]

; begin the program:

select-account

forever [
    cls
    print a-line
    print rejoin [
        {^/"r":  Read Email^/}
        {^/"s":  Send Email^/}
        {^/"c":  Choose a different mail account^/}
        {^/"q":  Quit^/}
    ]
    print a-line
    response: ask "^/Select a menu choice:  "
    switch/default response [
        "r" [read-email]
        "s" [send-email]
        "c" [select-account]
        "q" [
            cls
            print "DONE!"
            wait .5
            quit
        ]
    ] [read-email]
]
```

9.2.2 Network Ports

One important use of ports is for transferring data via network connections (TCP and UDP "sockets"). When writing a network application, you must choose a specific port number through which data is to be transferred. Potential ports range from 0 to 65535, but many of those numbers are reserved for specific types of applications (email programs use port 110, web servers use port 80 by default, etc.). To avoid conflicting with other established network applications, it's best to choose a port number between 49152 and 65535 for small scripts. A list of reserved port numbers is available here.

Network applications are typically made up of two or more separate programs, each running on different computers. Any computer connected to a network or to the Internet is assigned a specific "IP address", notated in the format xxx.xxx.xxx.xxx. The numbers are different for every machine on a network, but most home and small business machines are normally in the IP range "192.168.xxx.xxx". You can obtain the IP address of your local computer with the following REBOL code:

```
read join dns:// (read dns://)
```

"Server" programs open a chosen network port and wait for one or more "client" programs to open the same port and then insert data into it. *The port opened by the server program is referred to in a client program by combining the IP address of the computer on which the server runs, along with the chosen port number, each separated by a colon symbol* (i.e., 192.168.1.2:55555).

The following simple set of scripts demonstrates how to use REBOL ports to transfer one line of text from a client to a server program. This example is intended to run on a single computer, for demonstration, so the word "localhost" is used to represent the IP address of the server (that's a standard convention used to refer to any computer's own local IP address). If you want to run this on two separate computers connected via a local area network, you'll need to obtain the IP address of the server machine (use the code above), and replace the word "localhost" with that number:

Here's the SERVER program. Be sure to run it *before* starting the client, or you will receive an error:

```
; Open network port 55555, in line mode (this mode expects full lines
; of text delineated by newline characters):

server: open/lines tcp://:55555

; Wait for a connection to the above port:

wait server

; Assign a label to the first connection made to the above port:

connection: first server

; Get the data which has been inserted into the above port object:

data: first connection

; Display the inserted data:

alert rejoin ["Text received: " data]

; Close the server

close server
```

Here's the CLIENT. Run it in a *separate* instance of the REBOL interpreter, after the above program has been started:

```
; Open the port created by the server above (replace "localhost" with
; an IP address if running these scripts on separate machines):

server-port: open/lines tcp://localhost:55555

; Insert some text into the port:

insert server-port "Hello Mr. Watson."

; Close the port:

close server-port
```

Typically, servers will continuously wait for data to appear in a port, and repeatedly do something with that data. The scripts below extend the above example with forever loops to continuously send, receive, and display messages transferred from client(s) to the server. This type of loop forms the basis for most peer-to-peer and client-server network applications. Type "end" in the client program below to quit both the client and server.

Here's the server program (run it first):

```
server: open/lines tcp://:55555             ; Open a TCP network port.
print "Server started...^/"
connection: first wait server               ; Label the first connection.
forever [
    data: first connection                  ; Get a line of data.
    print rejoin ["Text received: " data]   ; Display it.
    if find data "end" [
```

```
        close server                        ; End the program if the
        print "Server Closed"               ;   client user typed "end".
        halt
    ]
]
```

Here's the client program. Run it only *after the server program has been started*, and in a separate instance of the REBOL interpreter (or on a separate computer):

```
server-port: open/lines tcp://localhost:55555    ; Open the server port.
forever [
    user-text: ask "Enter some text to send:  "
    insert server-port user-text                 ; Transfer the data.
    if user-text = "end" [
        close server-port                        ; End the program if the
        print "Client Closed"                    ; user typed "end".
        halt
    ]
]
```

It's important to understand that REBOL servers like the one above can interact independently with more than one simultaneous client connection. The "connection" definition waits until a new client connects, and returns a port representing that first client connection. Once that occurs, "connection" refers to the port used to accept data transferred by the already connected client. If you want to add more simultaneous client connections during the forever loop, simply define another "first wait server". Try running the server below, then run two simultaneous instances of the above client:

```
server: open/lines tcp://:55555             ; Open a TCP network port.
print "Now start TWO clients..."
connection1: first wait server              ; Label the first connection.
connection2: first wait server              ; Label the second connection.
forever [
    data1: first connection1                ; Get a line of client1 data
    data2: first connection2                ; Get a line of client2 data
    print rejoin ["Client1: " data1]
    print rejoin ["Client2: " data2]
    if find data1 "end" [
        close server                        ; End the program if the
        print "Server Closed"               ;   client user typed "end".
        halt
    ]
]
```

Here's an example that demonstrates how to send data back and forth (both directions), between client and server. Again, run both programs in separate instances of the REBOL interpreter, and be sure to start the server first:

```
; Server:

print "Server started...^/"
port: first wait open/lines tcp://:55555
forever [
    user-text: ask "Enter some text to send:  "
    insert port user-text
    if user-text = "end" [close port  print "^/Server Closed^/"  halt]
    wait port
    print rejoin ["^/Client user typed:  " first port "^/"]
]
```

```
; Client:

port: open/lines tcp://localhost:55555
print "Client started...^/"
forever [
    user-text: ask "Enter some text to send:  "
    insert port user-text
    if user-text = "end" [close port  print "^/Client Closed^/"  halt]
    wait port
    print rejoin ["^/Server user typed:  " first port "^/"]
]
```

The following short script combines many of the techniques demonstrated so far. It can act as either server or client, and can send messages (one at a time), back and forth between the server and client:

```
do [
    either find ask "Server or client?  " "s" [
        port: first wait open/lines tcp://:55555 ; server
    ] [
        port: open/lines tcp://localhost:55555   ; client
    ]
    forever [
        insert port ask "Send:  "
        print join "Received: "first wait port
    ]
]
```

The following script is a complete GUI network instant message application. Unlike the FTP Chat Room presented earlier, the text in this application is sent directly between two computers, across a network socket connection (users of the FTP chat room never connect directly to one another - they simply connect to a universally available third party FTP server):

```
view layout [
    btn "Set client/server" [
        ip: request-text/title/default trim {
            Server IP (leave EMPTY to run as SERVER):
            } (to-string read join dns:// read dns://)
        either ip = "" [
            port: first wait open/lines tcp://:55555 z: true ; server
        ] [
            port: open/lines rejoin [tcp:// ip ":55555"] z: true
        ]
    ]
    r: area rate 4 feel [
        engage: func [f a e] [
            if a = 'time and value? 'z [
                if error? try [x: first wait port] [quit]
                r/text: rejoin [form x newline r/text] show r
            ]
        ]
    ]
    f: field "Type message here..."
    btn "Send" [insert port f/text]
]
```

Here's an even more compact version (probably the shortest instant messenger program you'll ever see!):

```
view layout [ across
    q: btn "Serve"[focus g p: first wait open/lines tcp://:8 z: 1]text"OR"
    k: btn "Connect"[focus g p: open/lines rejoin[tcp:// i/text ":8"]z: 1]
    i: field form read join dns:// read dns://  return
    r: area rate 4 feel [engage: func [f a e][if a = 'time and value? 'z [
        if error? try [x: first wait p] [quit]
        r/text: rejoin [x "^/" r/text] show r
    ]]]  return
    g: field "Type message here [ENTER]" [insert p value  focus face]
]
```

And here's an extended version of the above script that uploads your chosen user name, WAN/LAN IP, and port numbers to an FTP server, so that that info can be shared with others online (which enables them to find and connect with you). Connecting as server uploads the user info and starts the application in server mode. Once that is done, others can click the "Servers" button to retrieve and manually enter your connection info (IP address and port), to connect as client. By using different port numbers and user names, multiple users can connect to other multiple users, anywhere on the Internet:

```
server-list: ftp://username:password@yoursite.com/public_html/im.txt ;edit
view layout [ across
    q: btn "Serve" [
        parse read http://guitarz.org/ip.cgi[thru<title>copy p to</title>]
        parse p [thru "Your IP Address is: " copy pp to end]
        write/append server-list rejoin [
            b/text " " pp " " read join dns:// read dns://"  " j/text "^/"
        ]
        focus g p: first wait open/lines join tcp:// j/text z: 1
    ] text "OR"
    k: btn "Connect" [
        focus g p: open/lines rejoin [tcp:// i/text j/text] z: 1
    ]
    b: field 85 "Username"
    i: field 98 form read join dns:// read dns://
    j: field 48 ":8080" return
    r: area rate 4 feel [engage: func [f a e][if a = 'time and value? 'z [
        if error? try [x: first wait p] [quit]
        r/text: rejoin [x "^/" r/text] show r
    ]]]  return
    g: field "Type message here [ENTER]" [insert p value  focus face]
    tabs 181 tab btn "Servers" [print read server-list]
]
```

If you want to run scripts like these between computers connected to the Internet by broadband routers, you'll likely need to learn how to "forward" ports from your router to the IP address of the machine running your server program. In most situations where a router connects a local home/business network to the Internet, only the router device has an IP address which is visible on the Internet. The computers themselves are all assigned IP addresses that are *only accessible within the local area network*. Port forwarding allows you to send data coming to the IP address of the router (the IP which is visible on the Internet), on a unique port, to a specific computer inside the local area network. A full discussion of port forwarding is beyond the scope of this tutorial, but it's easy to learn how to do - just type "port forwarding" into Google. You'll need to learn how to forward ports on *your particular brand and model of router*.

With any client-server configuration, only the server machine needs to have an exposed IP address or an open router/firewall port. The client machine can be located behind a router or firewall, without any forwarded incoming ports.

Another option that enables network applications to work through routers is "VPN" software. Applications such as hamachi, comodo, and OpenVPN allow you to connect two separate LAN

networks across the Internet, and treat all the machines as if they are connected locally (connect to any computer in the VPN using a local IP address, such as 192.168.1.xxx). VPN software also typically adds a layer of security to the data sent back and forth between the connected machines. The down side of VPN software is that data transmission can be slower than direct connection using port forwarding (the data travels through a third party server).

9.2.3 Peer-to-Peer Instant Messenger

The following text message example contains line by documentation of various useful coding techniques. For instructions, see the help documentation included in the code.

```
REBOL [Title: "Peer-to-Peer Instant Messenger"]

; The following line sets a flag variable, used to mark whether or not
; the two machines have already connected.  It helps to more gracefully
; handle connection and shutdown actions throughout the script:

connected: false

; The code below traps the close button (just a variation of the routine
; used in the earlier listview example).  It assures that all open ports
; are closed, and sends a message to the remote machine that the
; connection has been terminated.  Notice that the lines in the disconnect
; message are sent in reverse order.  When they're received by the other
; machine, they're printed out one at a time, each line on top of the
; previous - so it appears correctly when viewed on the other side:

insert-event-func closedown: func [face event] [
    either event/type = 'close [
        if connected [
            insert port trim {
                ************************************************
                AND RECONNECT.
                YOU MUST RESTART THE APPLICATION
                TO CONTINUE WITH ANOTHER CHAT,
                THE REMOTE PARTY HAS DISCONNECTED.
                ************************************************
            }
            close port
            if mode/text = "Server Mode" [close listen]
        ]
        quit
    ] [event]
]

view/new center-face gui: layout [
    across
    at 5x2  ; this code positions the following items in the GUI

    ; The text below appears as a menu option in the upper
    ; left hand corner of the GUI.  When it's clicked, the
    ; text contained in the "display" area is saved to a
    ; user selected file:

     text bold "Save Chat" [
        filename: to-file request-file/title/file/save trim {
            Save file as:} "Save" %/c/chat.txt
        write filename display/text
    ]

    ; The text below is another menu option.  It displays
    ; the user's IP address when clicked.  It relies on a
```

```
; public web server to find the external address.
; The "parse" command is used to extract the IP address
; from the page.  Parse is covered in a separate
; dedicated section later in the tutorial.

text bold "Lookup IP" [
    parse read http://guitarz.org/ip.cgi [
        thru <title> copy my-ip to </title>
    ]
    parse my-ip [
        thru "Your IP Address is: " copy stripped-ip to end
    ]
    alert to-string rejoin [
        "External: " trim/all stripped-ip "  "
        "Internal: " read join dns:// read dns://
    ]
]

; The text below is a third menu option.  It displays
; the help text when clicked.

 text bold "Help" [
    alert {
    Enter the IP address and port number in the fields
    provided.  If you will listen for others to call you,
    use the rotary button to select "Server Mode" (you
    must have an exposed IP address and/or an open port
    to accept an incoming chat).  Select "Client Mode" if
    you will connect to another's chat server (you can do
    that even if you're behind an unconfigured firewall,
    router, etc.).  Click "Connect" to begin the chat.
    To test the application on one machine, open two
    instances of the chat application, leave the IP set
    to "localhost" on both.  Set one instance to run as
    server, and the other as client, then click connect.
    You can edit the chat text directly in the display
    area, and you can save the text to a local file.
    }
]
return

; Below are the widgets used to enter connection info.
; Notice the labels assigned to each item.  Later, the
; text contained in these widgets is referred to as
; <label>/text.  Take a good look at the action block
; for the rotary button too.  Whenever it's clicked,
; it either hides or shows the other widgets.  When in
; server mode, no connection IP address is needed - the
; application just waits for a connection on the given
; port.  Hiding the IP address field spares the user some
; confusion.

lab1: h3 "IP Address:"  IP: field "localhost" 102
lab2: h3 "Port:" portspec: field "9083" 50
mode: rotary 120 "Client Mode" "Server Mode" [
    either value = "Client Mode" [
        show lab1 show IP
    ][
        hide lab1 hide IP
    ]
]

; Below is the connect button, and the large action block
```

```
; that does most of the work.  When the button is clicked,
; it's first hidden, so that the user isn't tempted to
; open the port again (that would cause an error).  Then,
; a TCP/IP port is opened - the type (server/client) is
; determined using an "either" construct.  If an error
; occurs in either of the port opening operations, the
; error is trapped and the user is alerted with a message -
; that's more graceful and informative than letting the
; program crash with an error.  Notice that the IP
; address and port info are gathered from the fields above.
; If the server mode is selected (i.e., if the "mode" button
; above isn't displaying the text "Client Mode"), then the
; the TCP ports are opened in listening mode - waiting
; for a client to connect.  If the client mode is selected,
; an attempt is made to open a direct connection to the IP
; address and port selected.

cnnct: button red "Connect" [
    hide cnnct
    either mode/text = "Client Mode" [
        if error? try [
            port: open/direct/lines/no-wait to-url rejoin [
                "tcp://" IP/text ":" portspec/text]
        ][alert "Server is not responding." return]
    ][
        if error? try [
            listen: open/direct/lines/no-wait to-url rejoin [
                "tcp://:" portspec/text]
            wait listen
            port: first listen
        ][alert "Server is already running." return]
    ]

    ; After the ports have been opened, the text entry field
    ; is highlighted, and the connection flag is set to true.
    ; Focusing on the text entry field provides a nice visual
    ; cue to the user that the connection has been made, but
    ; it's not required.

    focus entry
    connected: true

    ; The forever loop below continuously waits for data to
    ; appear in the open network connection.  Whenever data
    ; is inserted on the other side, it's copied and
    ; appended to the current text in the display area, and
    ; then the display area is updated to show the new text.

    forever [
        wait port
        foreach msg any [copy port []] [
            display/text: rejoin [
                ">>>  "msg newline display/text]
        ]
        show display
    ]
]

; Below are the display area and text entry fields.  Notice
; the labels assigned to each.  The "return"s just put each
; widget on a new line in the GUI (because the layout mode
; is set to "across" above).
```

```
    return  display: area "" 537x500
    return  entry: field 428  ; the numbers are pixel sizes

    ; The send button below does some more important work.
    ; First, it checks to see if the connection has been made
    ; (using the flag set above).  If so, it inserts the text
    ; contained in the "entry" field above into the open TCP/IP
    ; port, to be picked up by the remote machine - if the
    ; connection has been made, the program on the other end
    ; is waiting to read any data inserted into that port.
    ; After sending the data across the network connection,
    ; the text is appended to the local current text display
    ; area, and the display is updated:

    button "Send Text" [
        if connected [
            insert port entry/text focus entry
            display/text: rejoin [
                "<<<  " entry/text newline display/text]
            show display
        ]
    ]
]

show gui do-events  ; these are required because the "/new"
                    ; refinement is used above.
```

9.2.4 Transferring Binary Files Through TCP Network Sockets:

These 2 scripts based on http://www.rebol.net/cookbook/recipes/0058.html, by Carl Sassenrath (edited and condensed here), demonstrate how to transfer binary files directly between any two networked computers (across a TCP socket connection), using ports. Sending binary files is different from sending text in that the length of the file must be transmitted before sending the file. That must be done so that the receiving code knows when the complete file has been transmitted. The sending script below appends the file length information, and the file name, to the data being sent. The receiving script searches for that information, then receives the specified amount of binary data and saves it to a file when complete:

```
REBOL [Title: "Server/Receiver"]

p: ":8000"  ; port #
print "receiving"

data: copy wait client: first port: wait open/binary/no-wait join tcp:// p
info: load to-string copy/part data start: find data #""
remove/part data next start
while [info/2 > length? data] [append data copy client]
write/binary (to-file join "transferred-" (second split-path info/1)) data

insert client "done" wait client close client close port print "Done" halt


REBOL [Title: "Client/Sender"]

ip: "localhost" p: ":8000"  ; IP address and port #
print "sending"

data: read/binary file: to-file request-file
server: open/binary/no-wait rejoin [tcp:// ip p]
insert data append remold [file length? data] #""
insert server data
```

```
wait server close server print "Done" halt
```

Here's a more compact example that demonstrates how to create and send an image from one computer to another:

```
; server/receiver - run first:

if error? try [port: first wait open/binary/no-wait tcp://:8] [quit]
mark: find file: copy wait port #""
length: to-integer to-string copy/part file mark
while [length > length? remove/part file next mark] [append file port]

view layout [image load file]

; client/sender - run after server (change IP address if using on 2 pcs):

save/png %image.png to-image layout [box blue "I traveled through ports!"]

port: open/binary/no-wait tcp://127.0.0.1:8  ; adjust this IP address
insert file: read/binary %image.png join l: length? file #""
insert port file
```

The following program is a walkie-talkie push-to-talk type of voice over IP application. It's extremely simple - it just records sound from mic to .wav file, then transfers the wave file to another IP (where the same program is running), for playback. Sender and receiver open in separate processes, and both run in forever loops to enable continuous communication back and forth. As it stands, this is a MS Windows only application. The code which handles the sound recording is discussed in more detail in the section of this tutorial about DLLs:

```
REBOL [Title: "Intercom (VOIP Messenger)"]

write %wt-receiver.r {
    REBOL []
    print join "Receiving at " read join dns:// read dns://
    if error? try[c: first t: wait open/binary/no-wait tcp://:8000][quit]
    s: open sound://
    forever [
        d: copy wait c
        if error? try [i: load to-string copy/part d start: find d #""] [
            print "^lclient closed" close t close c close s wait 1 quit
        ]
        remove/part d next start
        while [i/2 > length? d] [append d copy c]
        write/binary (to-file join "t-" (second split-path i/1))
            decompress to-binary d
        insert s load %t-r.wav wait s
    ]
}
launch %wt-receiver.r

lib: load/library %winmm.dll
mciExecute: make routine! [c [string!] return: [logic!]] lib "mciExecute"

if (ip: ask "Connect to IP (none = localhost):  ") = "" [ip: "localhost"]
if error? try [s: open/binary/no-wait rejoin [tcp:// ip ":8000"]] [quit]

mciExecute "open new type waveaudio alias buffer1 buffer 4"
forever [
```

```
    x: ask "^lPress [ENTER] to start sending sound (or 'q' to quit): "
    if find x "q" [close s  free lib  break]
    ; if (ask "^lPress [ENTER] to send sound ('q' to quit): ") = "q"[quit]
    mciExecute "record buffer1"
    ask "^l*** YOU ARE NOW RECORDING SOUND ***   Press [ENTER] to send:  "
    mciExecute join "save buffer1 " to-local-file %r.wav
    mciExecute "delete buffer1 from 0"
    data: compress to-string read/binary %r.wav
    insert data append remold [%r.wav length? data] #""
    insert s data
]
```

Here's a more compact version of the above application, with hands-free operation enabled (several obfuscated versions of this script can be found at the end of this tutorial - likely the most compact VOIP programs you'll find anywhere):

```
REBOL [title: "VOIP"] do [write %ireceive.r {REBOL []
if error? try [port: first wait open/binary/no-wait tcp://:8] [quit]
wait 0  speakers: open sound://
forever [
    if error? try [mark: find wav: copy wait port #""] [quit]
    i: to-integer to-string copy/part wav mark
    while [i > length? remove/part wav next mark] [append wav port]
    insert speakers load to-binary decompress wav
]} launch %ireceive.r
lib: load/library %winmm.dll
mci: make routine! [c [string!] return: [logic!]] lib "mciExecute"
if (ip: ask "Connect to IP (none = localhost):  ") = "" [ip: "localhost"]
if error? try [port: open/binary/no-wait rejoin [tcp:// ip ":8"]] [quit]
mci "open new type waveaudio alias wav"
forever [
    mci "record wav"  wait 2  mci "save wav r"  mci "delete wav from 0"
    insert wav: compress to-string read/binary %r join 1: length? wav #""
    if 1 > 4000 [insert port wav]  ; squelch (don't send) if too quiet
]]
```

For more information on ports, see http://www.rebol.com/docs/core23/rebolcore-14.html, http://stackoverflow.com/questions/1291127/rebol-smallest-http-server-in-the-world-why-first-wait-listen-port, and http://www.rebol.net/docs/async-ports.html.

9.3 Parse (REBOL's Answer to Regular Expressions)

The "parse" function is used to import and convert organized chunks of external data into the block format that REBOL recognizes natively. It also provides a means of dissecting, searching, comparing, extracting, and acting upon organized information within unformatted text data (similar to the pattern matching functionality implemented by regular expressions in other languages).

The basic format for parse is:

```
parse <data> <matching rules>
```

Parse has several modes of use. The simplest mode just splits up text at common delimiters and converts those pieces into a REBOL block. To do this, just specify "none" as the matching rule. Common delimiters are spaces, commas, tabs, semicolons, and newlines. Here are some examples:

```
text1: "apple orange pear"
parsed-block1: parse text1 none

text2: "apple,orange,pear"
parsed-block2: parse text2 none

text3: "apple        orange                    pear"
parsed-block3: parse text3 none

text4: "apple;orange;pear"
parsed-block4: parse text4 none

text5: "apple,orange pear"
parsed-block5: parse text5 none

text6: {"apple","orange","pear"}
parsed-block6: parse text6 none

text7: {
apple
orange
pear
}
parsed-block7: parse text7 none
```

To split files based on some character other than the common delimiters, you can specify the delimiter as a rule. Just put the delimiter in quotes:

```
text: "apple*orange*pear"
parsed-block: parse text "*"

text: "apple&orange&pear"
parsed-block: parse text "&"

text: "apple    &    orange&pear"
parsed-block: parse text "&"
```

You can also include mixed multiple characters to be used as delimiters:

```
text: "apple&orange*pear"
parsed-block: parse text "&*"
```

```
text: "apple&orange*pear"
parsed-block: parse text "*&" ; the order doesn't matter
```

Using the "splitting" mode of parse is a great way to get formatted tables of data into your REBOL programs. Splitting the text below by carriage returns, you run into a little problem:

```
text: {    First Name
           Last Name
           Street Address
           City, State, Zip}

parsed-block: parse text "^/"

; ^/ is the REBOL symbol for a carriage return
```

Spaces are included in the parsing rule by default (parse automatically splits at all empty space), so you get a block of data that's more broken up than intended:

```
["First" "Name" "Last" "Name" "Street" "Address" "City,"
    "State," "Zip"]
```

You can use the "/all" refinement to eliminate spaces from the delimiter rule. The code below:

```
text: {    First Name
           Last Name
           Street Address
           City, State, Zip}

parsed-block: parse/all text "^/"
```

converts the given text to the following block:

```
[" First Name" "      Last Name" "      Street Address"
    "      City, State, Zip"]
```

Now you can trim the extra space from each of the strings:

```
foreach item parsed-block [trim item]
```

and you get the following parsed-block, as intended:

```
["First Name" "Last Name" "Street Address" "City, State, Zip"]
```

Pattern Matching Mode:

You can use parse to check whether any specific data exists within a given block. To do that, specify the rule (matching pattern) as the item you're searching for. Here's an example:

```
parse ["apple"] ["apple"]
```

```
parse ["apple" "orange"] ["apple" "orange"]
```

Both lines above evaluate to true because they match exactly. IMPORTANT: By default, as soon as parse comes across something that doesn't match, the entire expression evaluates to false, EVEN if the given rule IS found one or more times in the data. For example, the following is false:

```
parse ["apple" "orange"] ["apple"]
```

But that's just default behavior. You can control how parse responds to items that don't match. Adding the words below to a rule will return true if the given rule matches the data in the specified way:

1. "any" - the rule matches the data zero or more times
2. "some" - the rule matches the data one or more times
3. "opt" - the rule matches the data zero or one time
4. "one" - the rule matches the data exactly one time
5. an integer - the rule matches the data the given number of times
6. two integers - the rule matches the data a number of times included in the range between the two integers

The following examples are all true:

```
parse ["apple" "orange"] [any string!]
parse ["apple" "orange"] [some string!]
parse ["apple" "orange"] [1 2 string!]
```

You can create rules that include multiple match options - just separate the choices by a "|" character and enclose them in brackets. The following is true:

```
parse ["apple" "orange"] [any [string! | url! | number!]]
```

You can trigger actions to occur whenever a rule is matched. Just enclose the action(s) in parentheses:

```
parse ["apple" "orange"] [any [string!
    (alert "The block contains a string.") | url! | number!]]
```

You can skip through data, ignoring chunks until you get to, or past a given condition. The word "to" ignores data UNTIL the condition is found. The word "thru" ignores data until JUST PAST the condition is found. The following is true:

```
parse [234.1 $50 http://rebol.com "apple"] [thru string!]
```

The real value of pattern matching is that you can search for and extract data from unformatted text, in an organized way. The word "copy" is used to assign a variable to matched data. For example, the following code downloads the raw HTML from the REBOL homepage, ignores everything except what's between the HTML title tags, and displays that text:

```
parse read http://rebol.com [
    thru <title> copy parsed-text to </title> (alert parsed-text)
]
```

The following code extends the example above to provide the useful feature of displaying the external ip address of the local computer. It reads http://guitarz.org/ip.cgi, parses out the title text, and then parses that text again to return only the IP number. The local network address is also displayed, using the built in dns protocol in REBOL:

```
parse read http://guitarz.org/ip.cgi [
    thru <title> copy my-ip to </title>
]
parse my-ip [
    thru "Your IP Address is: " copy stripped-ip to end
]
alert to-string rejoin [
    "External: " trim/all stripped-ip "  "
    "Internal: " read join dns:// read dns://
]
```

Here's a useful example that removes all comments from a given REBOL script (any part of a line that begins with a semicolon ";"):

```
code: read to-file request-file

parse/all code [any [
    to #";" begin: thru newline ending: (
        remove/part begin ((index? ending) - (index? begin))) :begin
    ]
]

editor code
```

For more about parse, see the following links:

http://www.codeconscious.com/rebol/parse-tutorial.html
http://www.rebol.com/docs/core23/rebolcore-15.html
http://en.wikibooks.org/wiki/REBOL_Programming/Language_Features/Parse
http://www.rebolforces.com/zine/rzine-1-06.html#sect4.

9.4 2D Drawing, Graphics, and Animation

With REBOL's "view layout" ("VID") dialect you can easily build graphic user interfaces that include buttons, fields, text lists, images and other GUI widgets, but it's not meant to handle general purpose graphics or animation. For that purpose, REBOL includes a built-in "draw" dialect. Various drawing functions allow you to make lines, boxes, circles, arrows, and virtually any other shape. Fill patterns, color gradients, and effects of all sorts can be easily applied to drawings.

Implementing draw functions typically involves creating a 'view layout' GUI, with a box widget that's used as the viewing screen. "Effect" and "draw" functions are then added to the box definition, and a block is passed to the draw function which contains more functions that actually perform the drawing of various shapes and other graphic elements in the box. Each draw function takes an appropriate set of arguments for the type of shape created (coordinate values, size value, etc.). Here's a basic example of the draw format:

```
view layout [box 400x400 effect [draw [line 10x39 322x211]]]
;  "line" is a draw function
```

Here's the exact same example indented and broken apart onto several lines:

```
view layout [
    box 400x400 effect [
        draw [
            line 10x39 322x211
        ]
    ]
]
```

Any number of shape elements (functions) can be included in the draw block:

```
view layout [
    box 400x400 black effect [
        draw [
            line 0x400 400x50
            circle 250x250 100
            box 100x20 300x380
            curve 50x50 300x50 50x300 300x300
            spline closed 3 20x20 200x70 150x200
            polygon 20x20 200x70 150x200 50x300
        ]
    ]
]
```

Color can be added to graphics using the "pen" function. Shapes can be filled with color, with images, and with other graphic elements using the "fill-pen" function. The thickness of drawn lines is set with the "line-width" function:

```
view layout [
    box 400x400 black effect [
        draw [
            pen red
            line 0x400 400x50
            pen white
            box 100x20 300x380
            fill-pen green
            circle 250x250 100
```

```
            pen blue
            fill-pen orange
            line-width 5
            spline closed 3 20x20 200x70 150x200
            polygon 20x20 200x70 150x200 50x300
        ]
    ]
]
```

Gradients and other effects can be easily applied to the elements:

```
view layout [
    box 400x220 effect [
        draw [
            fill-pen 200.100.90
            polygon 20x40 200x20 380x40 200x80
            fill-pen 200.130.110
            polygon 20x40 200x80 200x200 20x100
            fill-pen 100.80.50
            polygon 200x80 380x40 380x100 200x200
        ]
        gradmul 180.180.210 60.60.90
    ]
]
```

Drawn shapes are automatically anti-aliased (lines are smoothed), but that default feature can be disabled:

```
view layout [
    box 400x400 black effect [
        draw [
            ;  with default smoothing:
            circle 150x150 100
            ;  without smoothing:
            anti-alias off
            circle 250x250 100
        ]
    ]
]
```

9.4.1 Animation

Animations can be created with draw by changing the coordinates of image elements. The fundamental process is as follows:

1. Assign a word label to the box in which the drawing takes place (the word "scrn" is used in the following examples).
2. Create a new draw block in which the characteristics of the graphic elements (position, size, etc.) are changed.
3. Assign the new block to "{yourlabel}/effect/draw" (i.e., "scrn/label/draw: [changed draw block]" in this case).
4. Display the changes with a "show {yourlabel}" function (i.e., "show scrn" in this case).

Here's a basic example that moves a circle to a new position when the button is pressed:

```
view layout [
    scrn: box 400x400 black effect [draw [circle 200x200 20]]
```

```
    btn "Move" [
        scrn/effect/draw: [circle 200x300 20]  ; replace the block above
        show scrn
    ]
]
```

Variables can be assigned to positions, sizes, and/or other characteristics of draw elements, and loops can be used to create smooth animations by adjusting those elements incrementally:

```
pos: 200x50
view layout [
    scrn: box 400x400 black effect [draw [circle pos 20]]
    btn "Move Smoothly" [
        loop 50 [
            ; increment the "y" value of the coordinate:
            pos/y: pos/y + 1
            scrn/effect/draw: copy [circle pos 20]
            show scrn
        ]
    ]
]
```

Animation coordinates (and other draw properties) can also be stored in blocks:

```
pos: 200x200
coords: [70x346 368x99 143x45 80x125 237x298 200x200]

view layout [
    scrn: box 400x400 black effect [draw [circle pos 20]]
    btn "Jump Around" [
        foreach coord coords [
            scrn/effect/draw: copy [circle coord 20]
            show scrn
            wait 1
        ]
    ]
]
```

Other data sources can also serve to control movement. In the next example, user data input moves the circle around the screen. Notice the use of the "feel" function to update the screen every 10th of a second ("rate 0:0:0.1"). Since feel is used to watch, wait for, and respond to window events, you'll likely need it in many situations where animation is used, such as in games:

```
pos: 200x200
view layout [
    scrn: box 400x400 black rate 0:0:0.1 feel [
        engage: func [face action event] [
            if action = 'time [
                scrn/effect/draw: copy []
                append scrn/effect/draw [circle pos 20]
                show scrn
            ]
        ]
    ] effect [ draw [] ]
    across
    btn "Up" [pos/y: pos/y - 10]
    btn "Down" [pos/y: pos/y + 10]
    btn "Right" [pos/x: pos/x + 10]
```

```
    btn "Left" [pos/x: pos/x - 10]
]
```

Here's a very simple paint program that also uses the feel function. Whenever a mouse-down action is detected, the coordinate of the mouse event ("event/offset") is added to the draw block (i.e., a new dot is added to the screen wherever the mouse is clicked), and then the block is shown:

```
view layout [
    scrn: box black 400x400 feel [
        engage: func [face action event] [
            if find [down over] action [
                append scrn/effect/draw event/offset
                show scrn
            ]
            if action = 'up [append scrn/effect/draw 'line]
        ]
    ] effect [draw [line]]
]
```

A useful feature of draw is the ability to easily scale and distort images simply by indicating 4 coordinate points. The image will be altered to fit into the space marked by those four points:

```
view layout [
    box 400x400 black effect [
        draw [
            image logo.gif 10x10 350x200 250x300 50x300
            ; "logo.gif" is built into the REBOL interpreter
        ]
    ]
]
```

Here's an example that incorporates the image scaling technique above with some animation. IMPORTANT: In the following example, the coordinate position calculations occur *inside the draw block*. Whenever such evaluations occur inside a draw block (i.e., when values are added or subtracted to a variable coordinate position, size, etc.), a *"reduce" or "compose"* function must be used to evaluate those values. Notice the tick mark (') next to the "image" function. Function words inside a reduced block need to be marked with that symbol to evaluate correctly:

```
pos: 300x300
view layout [
    scrn: box pos black effect [
        draw [image logo.gif 0x0 300x0 300x300 0x300]
    ]
    btn "Animate" [
        for point 1 140 1 [
            scrn/effect/draw: copy reduce [
                'image logo.gif
                (pos - 300x300)
                (1x1 + (as-pair 300 point))
                (pos - (as-pair 1 point))
                (pos - 300x0)
            ]
            show scrn
        ]
        for point 1 300 1 [
            scrn/effect/draw: copy reduce [
                'image logo.gif
```

```
                (1x1 + (as-pair 1 point))
                (pos - 0x300)
                (pos - 0x0)
                (pos - (as-pair point 1))
            ]
            show scrn
        ]
        ; no "reduce" is required below, because no calculations
        ; occur in the draw block - they're just static coords:
        scrn/effect/draw: copy [
            image logo.gif 0x0 300x0 300x300 0x300
        ]
        show scrn
    ]
]
```

Here's another example of a draw block which contains evaluated calculations, and therefore requires "reduce"d evaluation:

```
view layout [
    scrn: box 400x400 black effect [draw [line 0x0 400x400]]
    btn "Spin" [
        startpoint: 0x0
        endpoint: 400x400
        loop 400 [
            scrn/effect/draw: copy reduce [
                'line
                startpoint: startpoint + 0x1
                endpoint: endpoint - 0x1
            ]
            show scrn
        ]
    ]
]
```

The useful little paint program at http://rebol.org/cgi-bin/cgiwrap/rebol/view-script.r?script=paintplus.r consists of only 238 lines of code. Take a look at it to see how efficient REBOL's draw code is:

```
url: http://rebol.org/cgi-bin/cgiwrap/rebol/download-a-script.r?
script: "script-name=paintplus.r"
do rejoin [url script]
paint none []
```

For more information about built-in shapes, functions, and capabilities of draw, see http://www.rebol.com/docs/draw-ref.html, http://www.rebol.com/docs/draw.html, http://translate.google.com/translate?hl=en&sl=fr&u=http://www.rebolfrance.info/org/articles/login11/login11.htm (translated by Google), http://www.rebolforces.com/zine/rzine-1-05.html, http://www.rebolforces.com/zine/rzine-1-06.html (updated code for these two tutorials is available at http://mail.rebol.net/maillist/msgs/39100.html). A nice, short tutorial demonstrating how to build multi-player, networked games with draw graphics is available at RebolFrance (translated by Google). Also be sure to see http://www.nwlink.com/~ecotope1/reb/easy-draw.r (a clickable rebsite version is available in the REBOL Desktop -> Docs -> Easy Draw).

9.5 Using Animated GIF Images

Another easy way to work with animations in REBOL is with the "anim" style in GUIs. Anim takes a series of still image frames, and plays them in order as an animation with a given rate. The basic format is:

```
view layout [
    speed: 10
    anim rate (speed) [%image1.gif %image2.gif etc...]
]
```

The following script will convert an animated .gif into a folder filled with individual frame images:

```
REBOL []

gif-anim: load to-file request-file
make-dir %./frames/
count: 1

for count 1 length? gif-anim 1 [
    save/png rejoin [
        %./frames/ "your_file_name-" count ".png"
    ] pick gif-anim count
]
```

This next script will convert a directory of images (such as above, or any other series of images) into an embeddable block of REBOL code. It looks for all the images named [%your_file_name-1.* your_file_name-2.* etc...]:

```
REBOL []

system/options/binary-base: 64
file-list: read  %./frames/
anim-frames-block: copy []
foreach file file-list [
    ; Unique portion of file names for your image frames go here.
    ; Leave out this check if you instead want to convert all
    ; files in the directory:
    if find to-string file "your_file_name-" [
        print file
        uncompressed: read/binary file
        compressed: compress to-string uncompressed
        append anim-frames-block compressed
    ]
]

editor anim-frames-block
```

Here's some sample output:

```
anim-frames-block: [64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zCHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxqjufmZWdnWxS/OsPJcODoPLtKprUcW9TPLXJT
V7LdFZIZuMx/Ll/rrJCkC3NZ1ztd6SpVCG+L363EsXpCTvhmtovzVCWurr7R6jG7
rzZarKFpd8XTS77Z1/Xu7Qn+vunr6+/v725rqv6nm/Oj4Or2Ll7jvDUOa8+e6FX3
```

```
3uYjbPz0fN/RKjbeWcU+Z5do2qfN2lWaelnXfbveKwkz7ytLqu0qBK6Xed1cyfhG
TC58xeujhyuF422FXxQeOPybbR1nzbbP18+khtXvu/H95Ns7Gzdv5ZtfaVX64fjZ
crf/d6xPvV7XmJ7PZ1/x/ueXm/nXrOfVZKyZ+DL8nt85zhWzqu8LPosvPyYZEdW8
QrJjvjdj3TOFJuXQFVEVEl0iC9L49pVJJvZcnR7XLn/w+ux64XUpizrvbF0R1PFx
4QvB3s29OxLylB9tW9Cj9+vEol5NLk+5ia7vLB74GvxbETxZRklSqI+HyWNpR7ri
VbkJtreOp05nF1O/EeGW9C01/RqjmVrF3l7PZxnfPStv12qxsjBYAwBolvDW2AQA
AA==
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxqjufmZWdnW6hqBUwQfnxuvkPltxaJLSsuLOTt
ZWPdIPzSaal3vZUth6nWhZUsq7NsrUqzQ9f47K17qyWmdW1T2txFsreLdW/Pydu6
rXe2mHrsYuf3j86uLn95Z1/Qf6ZnWeUGD2e38V/3WVOh9viYkfzh3Fvmb1Iap+oq
P7OUKH64ocH2tsisGfkvTy7nXi6nG/n11dGZzLv3RQt8On3c19zY7e8stbyDCxtf
h0rLZBZuKjyYFrv6jsLdZ8xr99lGi3wueRLuGN6+zqSq7MW1700y/hHle4o/PhP8
5Xt+397f3z88Pj3ff/++v79/vGdnYbAGAJfEqNM/BAAA
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blri2cIVNC+GU2Hp6elcEX0tnsbLfPpNs++9mTE57fRcyepfJZxfFgUsdNWU
s51l8ihoma+8XatU6cOQVaHCca6zQh+GrYvlrWOVnvbgxrzUo/POzrz2JmpuLuu+
VuntT+9ML316T3VWuf79HXX/t/GuKTJIPBj5UW7bzB0fko75frwVGzP1ffIRa934
tpiQp88O9Zq3q84pL3qwq593uZ621dus61NCJ097K/714b7l3tf1bAv03jfNmv/v
264t3wu2Hn0r9973y6uiy2aql235hJeef35hovexONmK8jc3rzapXLeL03r+6cXl
1fHn9+39/f3D49Pz/ffv+/v7x+fX98/v3////1NWFgZrALxatNdHBAAA
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxKZMWCZWdnSuW+urOSId11nkP+rx6JLS8C2l0n
y6XO2PLyUovvXDtTCdNXV5pCl8YtnRn68tq6qOVNX6tKdW4uT+ud5sv9RTt6Xt79
Vz3a4Stu7Cq7+OitZ/i7i3tza5n4tCo+3JzWdniTz5oI1cfHNOVXt2pWqp87VaPv
LZf1413C3s7pdmKys0rSL88PZGbbe+vzva1rY3+/PV32+sCubRtnnd0rkJdwj/0h
0wyemh2p644UC7fl7H778NGh3vO6fKbGX1/f2Jx9/9ze3d/fPzjczSvvv2/Pz88v
Lq+Oj7dTYLAGANdbpyswBAAA
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxqjufmZWdnWxS/unNy8/Lz8x2auWR/BTVeXOwi
Khe7y2Sl47KAiVamXApZV5b4rnWSXbVVO3RB3OF/PN7X1G9usjnfdXdl2dpz2/IK
D339VZZ3fVfZ2kdnd5uqx++t+/9tqvaMlWfXh3IrT7sZ/jHxaHim0zWtSqOnM6a9
FDtbU26cfkDPvrlNc1dm6kVTb22Lv5alaYfm5C+qu3OrNPfa+tzj13Ijv+XemZzI
zv9n+oq7Kye6f9+js2Fz5IFZx4PK+MR+JSy/sTn7/rm9u7+/f3C4m/m7pACDNQAX
yZ/iJgQAAA==
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWendyiezhkdy8zHemsfm9O5LG6m7zHGqjWKRCMo7MY+h4
Z/IrYGXwMp65dq2rAl6FrGJbG3fUKuB12DrPvVqs2gFvwlelHZ/ku3qadvSilMP7
9kqW653fWvay6ezq67rxS6r/P1qjPWPDg4Nu/N+/rvyh9/iYt7zzNs0So6enpi2M
cuuRNLp3qJH/d6hNlEnY+eXS09l6w0qzLq+PPP7s98yy3N2Fp5+dvTtVN78lqf77
u5XTi3wfHpYVj5lTnX3xfsHkeDe98qrS11catc/PK7D+/u74fnNpHv19e35+fnF5
dfz5fXt/f//w+PR8/f//37H5mYGJisAYARqapGj4EAAA=
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8loBjYyMaj5ToqJNHjqOV0zsq/l56RnnjPlcq9t6Zy8+Nx8w+okFq8vywK6XDvl
ZGdNeR7Uyb9oUY7X55dH2INX7trCZbr62oIYSa+vv65mRDRHs05rrRR7GLU09+K+
v5LmD++sKuW/d3R2+YO4fbUn//G+MV+bsKpF9JzvnSKDx/vbhJ3DTkbo3j5coB2v
F72z4MzWubrBbLJWL25fWuZv7/d6y4q0bdMNj6udub7mzYnGuVV+v6qK8k/sl/We
l7Nb/+Ojyv5ytX0yFq/2LnRdfW3P79ef515b73/9nFRGSVPJ00c2fXwSf9685y1d
7B9ft/fu53ei/f3/5xnVtie8f33//P79wBaTeNBA4tYxoNGDrUVD5p4zF48aBZw
00h0ZGRksAYAd264o18EAAA=
} 64#{
```

```
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5tUvVi5Yia1eG5edqbPtPjSnpWBy/0YDCvDvvwsXh7Q6TL5
kI1UYGbQMv65Wq2nAl6FrApd++vIrA8HmRc4smbxni59cH294d46Vu2tOQc3OzDO
cc2+ujZiZ9zjc6mvr+hFNGV+/rT31bUX9xuTTybFWllsTFzXI5uv6xO2yXe3m669
nrfIxrAzDaLqx9bc2Jx8aVZ90bWcWYZXr6xj39+W++NT4K1VuZ9LeqPfpM2cWHj8
ytmQHx/u79b9zSf3e9un5iOth/QkYnd9fHVy/fSydbWl5e8PBbYHLreJ+1Oyv1d1
cX5tVe2Li+94t/X7y9b9Wf5y4mx3u5919d/Orr1+s8jyovr9ZFYpjol1XGYvHjQL
uGk8bBEJy3jYKpG24mGbTNmLh+0KbRqPOoTYWBisAbfrxM90BAAA
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxqjufmZWdnWxRHhRwIfu46z6Hx1xSJLSsuLOTt
1XLdFfDy0mIfTqu5t4xfOayKWMt04NRVretrAvc3yWqVrTm/LnqlUuusba9Ct6aL
ctQ4mL+9syt3+jHWgO+Nd/fVPXxm88p8Q8y+Gl7/q5Il667sZjp7S0drqm7UHP/T
UrJ7LNc/2zFFOXudlNWyG9uzvs6yO1NgEj29V3RXH2/1tzfTthVv9lt52+zdvcXZ
zPZ/rb99OKfvLF+vu+d50Xaju3b3bSutnj+fsTx4/sra6pK3N9fed2Op/2uR/OZ5
+/pQf7GKiJ37tlb905I3LVw7s//St1W7NgW8f/l1+41qZr6O+MxvjuH3m3jMXjxo
FnDTeNgiEpbxsFUibUViGyMjgzUAhlm/D2kEAAA=
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJGcFnIAgdVr2kGybtEJDernZmpnfsqp9P48bn5tvr/ZKSuPApY4Koo
Fzvry8OgZb6Sdq1Sog9DZjJlh/l6mLz2ZeDfU3c3SuClwzQm+RWsC6bqOC7JOrwo
Vnv72uht1gfbeK0n6MWtKW/8pbrj2/uI7QU/F9Vmf14XMbfnolxpjWlR3GGbyXZb
a3ZufLY619b5H8+vnNRL8z7K6ciWbnG80B7Y3SZrrZF7bVN+ee6q6uKr9/ZFM8/X
qfnx7s6xYPGrs+7oPXrWzex83qes6svaa+v/n9OrtUp9fX9ve7j/ux8fP3x61rjY
vLZ6b+iNdzsPre/9l5a86itjv21cXGXk5p+Wx+fVM3K9CK15v7MtwZlL74RCAp+b
xsMWkbCMh60SaSsetsmUvXjYrtCm8ahDZVrGo06NPFEBBmsAOJHArHoEAAA=
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4a8uKBYvd+6Wd
i/54bFp8YjKf9yqTzk2ph6ZqxZ4S4dj87Mw00+J7IjM3Pz/Xa1v674jElecXJrom
yq3NKFbwWC4/PSiE68FB5llMay/1aJkuClobLhqyV2pa9vUp8SeZBLjL1t7czDM7
S9ViukrMlpCNYj2V5YlB03x/7/uzu3RpQqsjL5tdjYFhyIF8yfehWT82Rmz3VxXf
9rvi0+VJs8zdv8lsLYo/NK2b699pqS93r20wLu/lrTbNvbYt3/rcWmv9x5f2prb7
1VZbvHxwrPO1n94u8+IzB/XV+/VsTEpfXl5pn+9Xbf3l6b2J1cHP+6psKhc/43zk
d99Cs/qrXW17eW3Nl7Jfp1aff17zb2/Rjz8/v8uWMf1aGt/IobbiQROP2YsHzQJu
Gg9bRMIyHrZKpK142CZT9uJhu0KbxqMOlWk7Eh0YrAGyBMCKdgQAAA==
} 64#{
eJxz93SzsEwMZwhn+M4AAg1g3ACmGsCsBjDnPxj/B1P/waz/YM4oGAXDBij+ZAHT
OiAClCfYOf4zMHLIeGxYcLCZQ1gr5sSGhYfbBZS95nhsXHS0W8I4686JjYuP9ys4
d8l4blpycrJG8KqYk5uWnp5ukHxqjufmZb79XEWvrlROfnRuvn21F4tXSOOFNptu
JttVBisuzfURtJsrdfXBleWhnHFLZ5VqX18V18lnImW6JmwT/yamD1ofHG9tZbi0
TLV6ytrbOwqeHkrNCtePaiypntX7u+z9rTml7OIxWiZrbhy2kbbm45IsTDrevTDu
GM/PgptrkzWj360qefhi9nLH+b09VUa3Z62zPN+zNkLt7fVt+eK21tHf8w40Jv7S
Oxv148Pxg73y1898t4h4Pnvh9rh5c9S+XjZbH/5+757K7y/22bc716+Lzn168ln4
db/1917kfwvbOH+6/zzLD8ez7p/X9/u1/d+fiEq2+Joe3owHjRxqKx408Zi9eNAs
4KbxsEUkLONhq0SaqACDNQAYMLy/ZgQAAA==
}]
```

And here's an example of how to write the files in that block back to the hard drive and display them in a GUI:

```
; Write files:

count: 1
make-dir %./frames/
for count 1 length? anim-frames-block 1 [
    write/binary rejoin [
        %./frames/ "frame-" count ".gif"
```

```
    ] to-binary decompress pick anim-frames-block count
]

; Create file list, with frames in numerical order:

file-list: read %./frames/
animation-frames: copy []
for count 1 length? file-list 1 [
    append animation-frames rejoin [
        %./frames/ "frame-" count ".gif"
    ]
]

; Display that file list as an animation:

view layout [
    anim: anim rate 10 frames animation-frames
]
```

Here's an example that combines the above animated GIF files with normal GUI animation:

```
view center-face layout [
    size 625x415
    backcolor black
    anim: anim rate 10 frames load animation-frames
    btn "Run Animation" [
        for counter 0 31 1 [
            anim/offset: anim/offset + (as-pair counter 0)
            show anim wait .05
        ]
        for counter 0 24 1 [
            anim/offset: anim/offset + (as-pair 0 counter)
            show anim wait .05
        ]
        for counter 0 31 1 [
            anim/offset: anim/offset + (as-pair (negate counter) 0)
            show anim wait .05
        ]
        for counter 0 24 1 [
            anim/offset: anim/offset + (as-pair 0 (negate counter))
            show anim wait .05
        ]
    ]
]
```

9.6 3D Graphics with r3D

The "r3D" modeling engine by Andrew Hoadley is built entirely from native REBOL 2D draw functions. It demonstrates the significantly powerful potential of draw. The examples below show some of what you can accomplish with r3D:

```
do http://www.rebol.net/demos/BF02D682713522AA/i-rebot.r
do http://www.rebol.net/demos/BF02D682713522AA/histogram.r
do http://www.rebol.net/demos/BF02D682713522AA/objective.r
```

The r3D engine is small. Here's the entire module in compressed, embeddable format (this is all just standard REBOL code compressed into a more compact format). To enable 3D graphics in your REBOL programs, just include this text in your code (paste it, or "do" it from a file). If you'd like to read and learn from the pure REBOL code that makes up this module, see the examples above (the r3D module is included in those examples as regular text code). The following example is available at http://re-bol.com/r3d.r:

```
do to-string decompress 64#{
eJzdPGtT28iWn+Nf0cOXsTMYkGXznL1bBMzEtQSnjPMAipqSpTboRpa8kmwwv37P
Od0tdethOzNTu1VLJUTqPu9XP5VR/8Pwmj002NhPA37KdmL7cqfBzhfpcxTD63no
xfyFfYwcL+Ar6PnK48SPwlNm7R3sHTQeG43GGbuI5qvYf3pOWdNtsc7BweEuMzER
6jOPZ36C2MxP2DOP+WTFnmInTLm3y6Yx5yyaMvfZiZ/4Lksj5oQrNgd+gBBNUscP
/fCJOcwFbgiZPgOZJJqmL07MGTBwQo85SRK5vgMkmRe5ixkPUydFllM/4Alrps+c
7dxKpJ0W8fG4EzA/ZNinutiLDyZYpCzmSRr7LtLYBR5+6AYLDwVRAIE/8yUPJEB2
SJDsIgE1UNhdNos8f4r/ctJtvpgEfvK8yzwfiU8WKTQm2OjyELCELvtRzBIeBEjD
B9lJ5VzCXdIX+MzRrqm0FHF+eY5mpjZgqekiDoEpJxwvAsshH+D6b+6m2IYI0ygI
ohdUz41Cz0etklN03hg6nUm05KSRcHYYpSCwEAN9Mc8dLLuSZwfkn3BpNuANZnY0
lWIUIEkhBnzwAfCZRzExLWq7R0J87LPb4dX42/mozwa37PNo+HVw2b+EOL2F951d
9m0w/jj8MmYAMTq/Gd+x4RU7v7lj/zW4udxl/e+fR/3bWzYcscGnz9eDPrQNbi6u
v1wObv5gHwAPmNwMx+x68GkwBrrjIfGU1Ab9W6T3qT+6+Aiv5x8G14Px3S67Goxv
kOwV0D1nn89H48HFl+vzEfv8ZfR5eNsHCS4F5ZvBzdUIePU/9W/Ge8Ab2lj/K7yw
24/n19fE7fwL6DAiKS+Gn+9Ggz8+jtnH4fVlHxo/9EG48w/XfcENVLu4Ph982mWX
55/O/wABgQ90DIHQiCCljN8+9qkJWJ7Dn4vxYHiDylwMb8YjeN0FXUfjDPXb4LYP
OTwa3KJlrkZD4IB2BQw03g0m/fDmpi/ooNVN5wAQvn+57efSXPbPr4HaLcqgA+9R
DWn/1R+qP1BDwpSCJra99syBrHrdi1mbiUeK0SXEOQRc4E9iJ141zpCrG3OoFZCH
0ynUoxDyYDWXqQaFKZlG8YxSO2k0kLDvAYyfrk5ZA2qm9gPVkB3kf40+bNP7S506
QGWnAsByi2Jgqs9BG38JJXu6CF0mpNm5EOo4TINQJsB6BT1LB6sAZvg88kFhjy1C
H0w3jWXNcB2oUs4OEfTYg8ddf+YEvzyyHQ/LCIJg2XJCyG9ZNYo021Bfwl9TqF4p
86m2vPE4Yr/8stN4NO1GLzH3FkDsQTeSVL/WkOq91l5rAZDKPmiH/z7SIIZmJZcH
YMAao6p+LHDCqMJK3w0rZVSgXZjWCSIwDRoKAvHVTwTWXQ3WXQlrpWHd12Ddl7De
JJa0eJWRv5eto7rv6g13vz5IH1WUJq4T1JmS+tYYUfRHYmyCaYLDpg5lL2Tm960M
uo7C3VbG3ZZCjaG/F2xTMtqdBlBp0ftNMW6YO45gFsK/GwY3jKoEhKrnBO6CwiYK
5QCegGKc1ElwiIYWN0ogsYMFTwgLmpG4eGDfqQ1ARKt4gNZSqCmRdfG155JqGll6
brYF51aNKTLRZJuOW4tQJ1CFPe8Me94V7fk3DXqXGfROM+hdZtA706Cyu6R81lKp
rdLMbCtCS0PftUqOuFtjyWqrmmVVGPLeMKSeaf+AHe8zO95rdrzP7HhfZcd7Xe/7
lqmJoW0OnZEuab7RPEU3bIWgno3IhDLE254fc1qVGHa9hAka/jxMgsj9gVXsK816
bIbTAViD4CCdoWIlmwY4lxf1bzFnBewvczVtmjkrMZGnZVlOgr+6fA6rDyeGeT0P
cLAHIQS9fSAD0/pXrI9sRb/f8LeqRPRySlO2EGdasCjiiE2dr1Wd+ObGUZLM4wh8
maLInkRYbYWAFhICNczZh2jdtzJS4lEIKZ+LgSFxOjlOR8PprMOxcxxbw7GrcbSY
2Fi85BwR4hQd1BYDmBEloulPBWL6/IwWaDZ5dMZTmEgyXEHa4HNYTePqDlwqgkIk
oCLzZxr9GUTRjz+dNKdmxJX+oyD0QEHXP2aBLFx5u5jALAccV8WmqMg/FVUQUiD0
/1ZUFbTYt346xooUOj8dcUUK9t+Nv1n3ddY1om5m5WEx61QGQMyTRZDmMyl8OxUb
Pg8CEFZk3HGf2YPDJsxl3iORzYR15nMOQ4TAzJ1gjnQOew8C7Fst9htrTsRLj15c
8XJCL54Es1s5biWhjk7oUCdkHRiUuq1qArZO4MggYBkEejUEujqBY4NAxyBwqBF4
NJR61EyuBhrw4dI2fZi7cLnGg9lcM+WzOTSZPiz6tc6xZb9Kcuscu9T8upSucemZ
rOzV66/mIGKNYu2/sCRiLzgVCXDAW8LiP5+WpPwZFoGghj+HXrE6wDKFVag5dRKo
m9LUfrhsv5wqkZti6SkVAcc1asNWwVggPRHZDNrZHtReAwoyqglTuojDqqhoO3Hs
rIzYWPIYN/2SQvGfddcFCiLx120yXkBqXDYlfSYriiCxH3Vfl7Qj9dxoBlWQt4vV
i9aqUP1TY6hk1ALzKBiNAg5+ZxbrvnYleHVtE1PUWbcdcJi9wa+n9Pk/NQYqEn3a
```

```
AoWxF8ffMGI8BAiKfpIb58RqQ0rnB2iSOPsPdqAniVJY28kyY98PfdyIxSGO9rKF
RWnniHaH/RgURDFWar+c5C25LtcFBrOiNjBIRjAnF2QyXRxJuKyCVaWCLDOG9BAp
jEIlFrTn6I5MbhVMRNlnnYyDPn6c5TmNiJApSz9aJIrfRDSH/BWkDTieLmiSCQNk
gx+b++4PPW4EY0lKyo7lpz7VaH+nMhq3GD7PWIYOcVoIykxalS0zqDOz/R78PcFn
SBqoJrP9Q3w5wF+Avo+tR/iCsBYCY+sxviCwdZitUaC0wMzxp+QGf4GfIFmxLGGV
97imS/SDjoywID9Fe3t75SWjQEcPizg9Vi6S01TsNhXGyopS56NA8YcUey+UXQd0
ICltINLdgN+r67fW45+w7fjbG/A38d+Ev0k/+7HstkpnZUGQO6zRbJIL94UsYp4l
WrrUcqS19KjluNVi7QyvU8KzS3iHCq9RQOtpQEclNKmfyc8qIR6XEE/K/OwaNF1w
6Q+TX7dGUAOxkyGWXJUROiwROqkjdGRI0CshSsNUyH5UJYIhhuDQKfnLLvmrW2H3
Tik+7FJ8IF6jgGeVDG+XDNGtiCurZHe7bAeTYbcGsVO2s8nRrpG0UzazybFXQpSC
VWDaBsvDEuZJnaw2sazI9LxAJxyPl7WkZzUlWmh6XFNWZN6rwKkG6mxBoq5uyiCv
6bXX4lpb8K2tt1vwXY+7XqNehXfKPmm6kTjxSFrFoiyn3PlYXVOlrW4pDa1eKQ9h
WVqRUN1S5huohxlqsVZbdimlrIqiVOBqlXGPy7gnFVztWsxuOa0KXLu1EnfLmUW4
m+q2Va5XlbR6FaXbqqhZlUr0NlZvaRSr7DG9ZEhZDqpkMUqLgitFk2UUuJM61JOy
cyyD6VFZXmWAssAnBtOjsnDKyOVR6MQMOqsMZ9cJfGAOV1ZZ4l4JVelgWvikXuKy
w0Diqmru4d7wzA/hmfrg/ZQ1Zf1RdUGMGHmkNGUFyvo7pX7b6LdL/V2jv9vK1664
xqd7Y8vIx+tfeNYhl3OcNnASusmRCStvAfBUrcDqNx9etVL3UL3h0ASg90iNtYob
DIWVXL4EVHsOXpTKnWRzR0XbMV3+zI6pEmlpkTuWarMT3jvivaPebfFut9TKTd/X
/uvSNKrkedAlAM+2dQk6rcqCooNYCsXKhK5FsTI92zpXq5XtdOobCXoFk3YQ+zKm
BX7eBcl/L5yYt+MoSiFCmkshl3KHkEo5Q6iJhsmcce555g0KaMCbKBGbgF4ObUVO
IClDPJvjPm6wiFW+K27XyXMaNlmoq02zBay2AMR9Fmd0jqnTpFJFP+Wz7TbnBaiV
7Wo5bOP2fFNslDgClUwhWiayxdifrtglUUdEpqWyg6OJuNvk/L81WPtnDfZJ7nGZ
BlOtQHiyYmGtuRx1oywz1y6jK73yRL/STKF+4P9/Zaj3LNxkmkva5zcNI9r+slnC
KGzT/be/aR8cwtbZyJ8y2vHdwz3fzzFexFOig+AkgZPi9n/KvR2lvSONAbTVgEjn
wf+o1VHwNXYXls9OV3+65OJG5wprYn5A6wh95d0FdNcqWrAXPwjYE4zQZvfEcX8Q
IdqPz2s/E1JL/zaZ2lFvaRZQjngwLmbJY4YKuGwkxJFgn0hWj2GkPIwPBLMOxJZk
6o82WD6ozTEs1mzKSotmh3FMnsYx8ziOyOi9mntl36PJsnCOuKxliatP7C7womZW
w+bsL1waxsu/hWvD4HQ/5HsxgxyHdzxG8MDxohnvCzdG1GJo8hLFgWcGqbhBa7bB
hOrf8sZMATZcOsmt/wYxgZPuyo1xmjyK2S0+dTz8oIAH4rcQACqPEz4FPFlXITIg
s1nIOxA79iIB8u172Wse0gKIvBcp1W9k7iKZiMiQILTbDLm4pzkF7a4CNLhRgN+d
5Nh2TvkTol+QNPopiym+YPKNpMpRz/Lr3OLCsjpEhBiKGLg6mTsunnfh7d5ldq9Z
qq6oEI2vgHrKCueggiueoOC/BTS8rgUWFyAdXSjs8cNpRJHo+GHCOoyCg0IyVAd+
u8xpEawscIkYf1r4SQeICdOTe4CfJjw1eA6AspD0Qi3arqD5RtLItBECbmEs5wlk
ZIvElxdftZjWTpYyS3W8sp3ySM6Ri6NJFqWELVKu441VazvxZ7igU6lA0mcK6wml
4kkvjGcwCs+htCdRLKxVTAzs2I+5CKY8q4zwV82aqHm9msIgg2wXEMeqHxbmmnln
zg9xvhot0gCvJyZ0Bw/vEdCHNVBz0mcnlEeX9LUMkmNFQxUPvH+FVhimcgFg6LX2
cJX/qHHH8U+nOnES7jHwIQlBn/5AuCQbmSGNNnLU+D02atB+VZao6c8M1dkEYG8C
6NaOg9lyuyoftrnKIEJtzQkmxdYSswsWy7BGXtpsmc7mFv2GV3fOAvy8LclvyFTV
6YxMzR0ZqgVSmIJnjYSlrzHEB18wAyRUH7LpVR3eI4kMf2mdiolbho7dUNNyiE4l
RO4yHOQrAOxGMfzyWpbjop1EwUjUyg0taBkQnSIEGDjnn4O689PyJT3NFWXIfO4I
TcUw3qc7QGo2ZQBIjVQlfgOPtaPYy/qlv09Z2zqAH+1iJ8zXaWvjX1lMPOTQ1PNo
wHZqYTslWLsW1jZgZQ7lcatQdItmuaQu1hSDPZvrlYv1LRVrI7my0YV7X7fOs6y+
bzN7gsJ/gYWfPaQL4F49p8KFkKrjSxGSIjkiKMKQIG60CEWovq3LVRgtIy/CVaHr
LmLcGFV1NB9wxEaprPlEXYhJj6esqanxHtYPPbmnOTWqkzSAnE9pJULNItTcggSH
xVzmvw1lw33mkLK+/KBW5CbnHn1JCstbXB1RI5TMAIqJnmfYLhJHJr4mspAjA1UC
S8A8eEwwKbww/m/MMljdn2octZEAZKdu9jved8pzU+iXDbeKZ91dJoosvPRHt5Hy
Flo2Trg2cwAz0FUgX36JYHDUI8uMlULWGSGohDOlV64TkUluMD+gEyrqlR/gzLpP
YhKBEuZSeqMqJQXLEkqV7NnCX6bOb8xJ2piPtMrFVay5UWtao+AuSqcshwxjYC5C
IRUPwOVgz9po12ZeDd5LxFYFklHhdQIZbGF9rW8aXUeO1x5eXZmX7KdTz0mdTZcf
E5mR4WL2lV7h4Uq19D0oxgQm83IxIxxwFXqV2lTFnr2yGSxE3nAm1MCth8Y7yAsp
Bbjh93+xX0FG6Hj3Tiyedz5QeNMWVQprCobd+J38jrZtQfOnh0z5d0rO04y0GEyh
XS50VLut2jlqkbd3649bloKysfMniGbzoHfZUKbG0mSGX56oFmqi2D1lPclozh1h
7NzKxT0s4X/hkPz+AdBIo7bcmGNNyhXlV+JRcwpa+JmtNtHBUrclrbctaHX0Kb80
huqz9TiGCIG4qZwpzNRtWYRZ1cCsNJi3Gpg3k9nvmcMAKvddgV0dlMmwDopY6s5X
2aNGv8w4KqOuongMGYWZJ2uiYVN9vieLLLVTndRTh+7q0uYirat25R1yHFDpaNLz
6PIrPGJdh1SUAkLSsqaKTk0YqHYdteH4riJihUL5+ZoFHv474QU/9TRQyH+SkhGm
+XV8sEohZpFOhXHKFVnfYNWKa1PFZTuLEjqk/Zm9Q9oyxEovZgaO2mti1IH/P8aI
T6CO+jilCZM0hiUHOJ3OnuhTYg9mUBPczXmOXnB/AUJl4sPIG/vgR7x1HkCcnQmi
iTpcWEAQ7bFbDhx5EL3AwIrRxl+dEPdAoikSw2iSXy1IZNy4RylHtpdvX56xmwi/
bcfGZDHH/2kD5vugkNrP2cuCWPy/B9AYR4sncfebtsVxnANX0jBHWzdnjQYq1Za7
fvKi9lm2AKTXvNY+0BGDZeyS5+3tqo72+p46WupvJY9iR3t9T94hA07oKEILJv4h
qIoj6QuuHl8gZtXExVCc9dihKQ68d4yWDrQcFVqOmG202NByXGg5Zl2jpQstvUJL
D/jpLcdA57DQcljAsoC/XWixM1606vsfGt6vyUFIAAA=
```

```
}
```

Here's a simple example that demonstrates the basic syntax and use of r3D. Be sure to do the code above before running this example:

```
Transx:  Transy:  Transz: 300.0          ; Set some camera
Lookatx:  Lookaty:  Lookatz: 100.0       ; positions to
                                         ; start with.
do update: does [            ; This "update" function is where
    world: copy []           ; everything is defined.
    append world reduce [    ; Add your 3D objects inside this "append".
        reduce [cube-model (r3d-scale 100.0 150.0 125.0) red]
    ]                        ; A red 'cube' 100x150x125 pixels is added.
    camera: r3d-position-object
        reduce [Transx Transy Transz]
        reduce [Lookatx Lookaty Lookatz]
        [0.0 0.0 1.0]
    RenderTriangles: render world camera r3d-perspective 250.0 400x360
    probe RenderTriangles    ; This line demonstrates what's going on
]                            ; under the hood.  You can eliminate it.

view layout [
    scrn: box 400x360 black effect [draw RenderTriangles]  ; basic draw
    across return
    slider 60x16 [Transx: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transy: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transz: (value * 600) update show scrn]
    slider 60x16 [Lookatx: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookaty: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookatz: (value * 200 ) update show scrn]
]
```

R3D works by rendering 3D images to native REBOL 2D draw functions, which are contained in the "RenderTriangles" block above. R3D provides basic shape structures and a simple language interface to create and view those images in a REBOL application. It automatically adjusts lighting and other characteristics of images as they're viewed from different perspectives. To see how the rendering of images is converted into simple REBOL draw functions, watch the output of the "probe RenderTriangles" line in the REBOL interpreter as you adjust the sliders above. It displays the list of draw commands used to create each image in the moving 3D world.

In the example above, slider widgets are used to adjust values in the animation. Those values could just as easily be controlled by loops or other forms of data input. In the example below, the values are adjusted by keystrokes assigned to empty text widgets (use the "asdfghqwerty" keys to move the cube):

```
Transx:  Transy:  Transz: 2.0
Lookatx:  Lookaty:  Lookatz: 1.0
do update: does [
    world: copy []
    append world reduce [
        reduce [cube-model (r3d-scale 100.0 150.0 125.0) red]
    ]
    Rendered: render world
        r3d-position-object
        reduce [Transx Transy Transz]
        reduce [Lookatx Lookaty Lookatz]
        [0.0 0.0 1.0]
        r3d-perspective 360.0 400x360
]
```

```
view layout [
    across
    text "" #"a" [Transx: (Transx + 10) update show scrn]
    text "" #"s" [Transx: (Transx - 10) update show scrn]
    text "" #"d" [Transy: (Transy + 10) update show scrn]
    text "" #"f" [Transy: (Transy - 10) update show scrn]
    text "" #"g" [Transz: (Transz + 10) update show scrn]
    text "" #"h" [Transz: (Transz - 10) update show scrn]
    text "" #"q" [Lookatx: (Lookatx + 10) update show scrn]
    text "" #"w" [Lookatx: (Lookatx - 10) update show scrn]
    text "" #"e" [Lookaty: (Lookaty + 10) update show scrn]
    text "" #"r" [Lookaty: (Lookaty - 10) update show scrn]
    text "" #"t" [Lookatz: (Lookatz + 10) update show scrn]
    text "" #"y" [Lookatz: (Lookatz - 10) update show scrn]
    at 20x20
    scrn: box 400x360 black effect [draw Rendered]
]
```

The r3D module can work with models saved in native .R3d format, and the "OFF" format (established by the GeomView program at http://www.geom.uiuc.edu/projects/visualization/. See http://local.wasp.uwa.edu.au/~pbourke/dataformats/oogl/#OFF for a description of the OFF file format). A number of OFF example objects are available at http://www.mpi-sb.mpg.de/~kettner/proj/obj3d/.

To understand how to create/import and manipulate more complex 3D shapes, examine the way objects are designed inside the "update" function in each of Andrew's three examples. Here's a simplified variation of Andrew's objective.r example that loads .off models from the hard drive. Be sure to do the r3D module code above before running this example, and then try downloading and loading some of the example .off files at the web site above:

```
RenderTriangles: []
view layout [
    scrn: box 400x360 black effect [draw RenderTriangles]
    across return
    slider 60x16 [Transx: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transy: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transz: (value * 600) update show scrn]
    slider 60x16 [Lookatx: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookaty: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookatz: (value * 200 ) update show scrn]
    return btn "Load Model" [
        model: r3d-load-OFF load to-file request-file
        modelsize: 1.0
        if model/3 [modelsize: model/3]
        if modelsize < 1.0 [ modelsize: 1.0 ]
        defaultScale: 200.0 / modelsize
        objectScaleX: objectScaleY: objectScaleZ: defaultscale
        objectRotateX: objectRotateY: objectRotateZ: 0.0
        objectTranslateX: objectTranslateY: objectTranslateZ: 0.0
        Transx:  Transy:  Transz: 300.0
        Lookatx:  Lookaty:  Lookatz: 200.0
        modelWorld: r3d-compose-m4 reduce [
            r3d-scale objectScaleX objectScaleY objectScaleZ
            r3d-translate
                objectTranslateX objectTranslateY objectTranslateZ
            r3d-rotatex objectRotateX
            r3d-rotatey objectRotateY
            r3d-rotatez objectRotateZ
        ]
        r3d-object: reduce [model modelWorld red]
        do update: does [
```

```
            world: copy []
            append world reduce [r3d-object]
            camera: r3d-position-object
                reduce [Transx Transy Transz]
                reduce [Lookatx Lookaty Lookatz]
                [0.0 0.0 1.0]
            RenderTriangles:
                render world camera r3d-perspective 250.0 400x360
        ]
        update show scrn
    ]
]
```

Like most REBOL solutions, r3D is a brilliantly simple, compact, and powerful design that doesn't require any external toolkits. It's pure REBOL, and it's really amazing!

9.6.1 Several 3D Scripts Using Raw REBOL Draw Dialect

The following short script is a compacted version of Gregory Pecheret's "ebuc-cube" (from http://www.rebol.net/demos/download.html). It demonstrates some simple 3d techniques using only native REBOL draw functions (no 3rd party library required). It's relatively easy to understand, manipulate, and use to create your own basic 3D graphics:

```
z: 10 h: z * 12 j: negate h c: as-pair z * 5 z * 5 l: z * 4 w: z * 20
img: to-image layout [box effect [draw [pen logo.gif circle c l]]]
q: make object! [x: 0 y: 0 z: 0]
cube: [[h h j] [h h h] [h j j] [h j h] [j h j] [j h h] [j j j] [j j h]]
view center-face layout [
    f: box 400x400 rate 0 feel [engage: func [f a e] [
        b: copy []   q/x: q/x + 5   q/y: q/y + 8   q/z: q/z + 3
        repeat n 8 [
            p: reduce pick cube n  ; point
            zx: (p/1 * cosine q/z) - (p/2 * sine q/z) - p/1
            zy: (p/1 * sine q/z) + (p/2 * cosine q/z) - p/2
            yx: (p/1 + zx * cosine q/y) - (p/3 * sine q/y) - p/1 - zx
            yz: (p/1 + zx * sine q/y) + (p/3 * cosine q/y) - p/3
            xy: (p/2 + zy * cosine q/x) - (p/3 + yz * sine q/x) - p/2 - zy
            append b as-pair (p/1 + yx + zx + w) (p/2 + zy + xy + w)
        ]
        f/effect: [draw [
            fill-pen 255.0.0.100  polygon b/6 b/2 b/4 b/8
            image img b/6 b/5 b/1 b/2
            fill-pen 255.159.215.100  polygon b/2 b/1 b/3 b/4
            fill-pen 54.232.255.100 polygon b/1 b/5 b/7 b/3
            fill-pen 0.0.255.100 polygon b/5 b/6 b/8 b/7
            fill-pen 248.255.54.100 polygon b/8 b/4 b/3 b/7
        ]]
        show f
    ]]
]
```

Here's a version that reshapes and moves the 3D cube in, out and around the screen:

```
g: 12 i: 5 h: i * g j: negate h w: 0 v2: v1: 1  ; sizes/positions
img: to-image layout [box 200.200.200.50 center logo.gif]
q: make object! [x: 0 y: 0 z: 0]
cube: [[h h j] [h h h] [h j j] [h j h] [j h j] [j h h] [j j j] [j j h]]
view center-face layout/tight [
    f: box 500x450 rate 0 feel [engage: func [f a e] [
```

```
        b: copy []  q/x: q/x + 3 q/y: q/y + 3 ; q/z: q/z + 3  ; spinning
        repeat n 8 [
            if w > 500 [v1: 0]   ; w: xy pos        v1: xy direction
            if w < 0 [v1: 1]
            either v1 = 1 [w: w + 1] [w: w - 1]
            if j > (g * i * 2) [v2: 0]  ; j: z pos (size) v2: z direction
            if j < g [v2: 1]
            either v2 = 1 [h: h - 1] [h: h + 1]  j: negate h
            p: reduce pick cube n  ; point
            zx: p/1 * cosine q/z - (p/2 * sine q/z) - p/1
            zy: p/1 * sine q/z + (p/2 * cosine q/z) - p/2
            yx: (p/1 + zx * cosine q/y) - (p/3 * sine q/y) - p/1 - zx
            yz: (p/1 + zx * sine q/y) + (p/3 * cosine q/y) - p/3
            xy: (p/2 + zy * cosine q/x) - (p/3 + yz * sine q/x) - p/2 - zy
            append b as-pair (p/1 + yx + zx + w) (p/2 + zy + xy + w)
        ]
        f/effect: [draw [
            image img b/6 b/5 b/1 b/2
            fill-pen 255.0.0.50      polygon b/6 b/2 b/4 b/8
            fill-pen 255.159.215.50  polygon b/2 b/1 b/3 b/4
            fill-pen 54.232.255.50   polygon b/1 b/5 b/7 b/3
            fill-pen 0.0.255.50      polygon b/5 b/6 b/8 b/7
            fill-pen 248.255.54.50   polygon b/8 b/4 b/3 b/7
        ]]
        show f
    ]]
]
```

And here's a little 3D game, with sound, based on the above code:

```
REBOL [title: "Little 3D Game"]

beep-sound: load to-binary decompress 64#{
eJwBUQKu/VJJRkZJAgAAV0FWRWZtdCAQAAAAAQABABErAAARKwAAAQAIAGRhdGEl
AgAA0d3f1cGadFQ+T2Z9jn1lSjM8T2uNsM/j7Midc05PWGh4eXVrXE5DQEZumsTn
4M2yk3hiVU9fcX+GcFU8KkNmj7rR3+HYroJbPUpfdoqAbldBP0ZWbpW62OvRrohk
WlleaHB2dW9bRzo1WYWy3OHbyrKObVNCVGp/jXpgRC48Vnievtfm6MCUaUVLWW1/
fXNkUkdCRlN7ps3r3cSkgm1fWFhmdH2AaVA6LElwnMja4dzNpHtXPUxje45/aVA5
PUtif6TG3uvMpHtXU1lkcnd2cGVURT0+ZJC84+HUvaGCZ1NIWm6AinVaQCtAX4Wu
yt3k37aJYEBKXXOHf3FdSEJET2KJsdPr1reUcGJbW2FsdXl2YUs5MFF7qdPe3tO+
mHNUP1Bnfo59ZEkyPFFukbTR5OvGm3BMTVlpent1aVpMQ0FJcZ3I6uHMsJB2YlZR
YXJ/hW5UOypEaJK90+Dg1qyBWjxKYHeLgG1WPz9HWXKYvNnr0KyFYVhZX2pydnVu
Wkc7N1yHtN3h2sivjGxTRFZrgI15X0MtPVh7osHZ5ua+kmdES1tvgn5zY1BGQ0hW
fqjO69vBoX9rXllaaHV9fmhPOi1Lcp/K2+DayaF4Vj1NY3uNfmhONjxLZIKnyODr
yqJ4VFFYZHN3dm5iUUM9QGaTv+Th0rqdf2VTS1tvgIl0WT4rQGCIssze5N60iF8/
Sl10h39vW0ZBRFFljLPU69W1kG1gWlxiYHkWb1ECAAA=
}
alert {
   Try to click the bouncing REBOLs as many times as possible in
   30 seconds.  The speed increases with each click!
}
do game: [
   speaker: open sound://
   g: 12 i: 5 h: i * g j: negate h  x: y: z: w: sc: 0  v2: v1: 1  o: now
   img1: to-image layout [backcolor brown box red center logo.gif]
   img2: to-image layout [backcolor aqua box yellow center logo.gif]
   img3: to-image layout [backcolor green box tan center logo.gif]
   cube: [[h h j][h h h][h j j][h j h][j h j][j h h][j j j][j j h]]
   view center-face layout/tight [
      f: box white 550x550 rate 15 feel [engage: func [f a e] [
         if a = 'time [
```

```
            b: copy []  x: x + 3  y: y + 3  ; z: z + 3
            repeat n 8 [
               if w > 500 [v1: 0]   if w < 50 [v1: 1]
               either v1 = 1 [w: w + 1] [w: w - 1]
               if j > (g * i * 1.4) [v2: 0]   if j < 1 [v2: 1]
               either v2 = 1 [h: h - 1] [h: h + 1]  j: negate h
               p: reduce pick cube n
               zx: p/1 * cosine z - (p/2 * sine z) - p/1
               zy: p/1 * sine z + (p/2 * cosine z) - p/2
               yx: (p/1 + zx * cosine y) - (p/3 * sine y) - p/1 - zx
               yz: (p/1 + zx * sine y) + (p/3 * cosine y) - p/3
               xy: (p/2 + zy * cosine x) - (p/3 + yz * sine x) - p/2 - zy
               append b as-pair (p/1 + yx + zx + w) (p/2 + zy + xy + w)
            ]
            f/effect: [draw [
               image img1 b/6 b/2 b/4 b/8
               image img2 b/6 b/5 b/1 b/2
               image img3 b/1 b/5 b/7 b/3
            ]]
            show f
            if now/time - o/time > :00:20 [
               close speaker
               either true = request [
                  join "Time's Up! Final Score: " sc "Again" "Quit"
               ] [do game] [quit]
            ]
         ]
         if a = 'down [
            xblock: copy [] yblock: copy []
            repeat n 8 [
                append xblock first pick b n
                append yblock second pick b n
            ]
            if all [
                e/offset/1 >= first minimum-of xblock
                e/offset/1 <= first maximum-of xblock
                e/offset/2 >= first minimum-of yblock
                e/offset/2 <= first maximum-of yblock
            ][
               insert speaker beep-sound wait speaker
               sc: sc + 1
               t1/text: join "Score: " sc
               show t1
               if (modulo sc 3) = 0 [f/rate: f/rate + 1]
               show f
            ]
         ]
      ]]
      at 200x0 t1: text brown "Click the bouncing REBOLs!"
   ]
]
```

9.7 Multitasking

"Threads" are a feature of modern operating systems that allow multiple pieces of code to run concurrently, without waiting for the others to complete. Without threads, individual portions of code must be evaluated in consecutive order. Unfortunately, REBOL does not implement a formal mechanism for threading at the OS level, but *does* contain built-in support for asynchronous network port and services activity. See http://www.rebol.net/docs/async-ports.html, http://www.rebol.net/docs/async-examples.html, http://www.rebol.net/rebservices/services-start.html, and http://www.rebol.net/rebservices/quick-start.html for more information.

The following technique provides an alternate way to evaluate other types of code in a multitasking manner:

1. Assign a rate of 0 to a GUI item in a 'view layout' block.
2. Assign a "feel" detection to that item, and put the actions you want performed simultaneously inside the block that gets evaluated every time a 'time event occurs.
3. Stop and start the evaluation of concurrently active portions of code by assigning a rate of "none" or 0, respectively, to the associated GUI item.

The following is an example of a webcam viewer which creates a video stream by repeatedly downloading and displaying images from a given webcam URL. To create a moving video effect, the process of downloading each image must run without stopping (i.e., in some sort of unending "forever" loop). But for a user to control the stop/start of the video flow (by clicking a button, for example), the interpreter must be able to check for user events that occur outside the forever loop. By running the repeated download using the technique outlined above, the program can continue to respond to other events while continuously looping the download code:

```
webcam-url: http://209.165.153.2/axis-cgi/jpg/image.cgi
view layout [
    btn "Start Video" [
        webcam/rate: 0
        webcam/image: load webcam-url
        show webcam
    ]
    btn "Stop Video" [webcam/rate: none show webcam]
    return
    webcam: image load webcam-url 320x240 rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/image: load webcam-url show face
            ]
        ]
    ]
]
```

Here's an example in which two webcam video updates are treated as separate processes. Both can be stopped and started as needed:

```
webcam-url: http://209.165.153.2/axis-cgi/jpg/image.cgi
view layout [
    across
    btn "Start Camera 1" [
        webcam/rate: 0
        webcam/image: load webcam-url
        show webcam
    ]
    btn "Stop Camera 1" [webcam/rate: none show webcam]
    btn "Start Camera 2" [
        webcam2/rate: 0
```

```
        webcam2/image: load webcam-url
        show webcam2
    ]
    btn "Stop Camera 2" [webcam2/rate: none show webcam2]
    return
    webcam: image load webcam-url 320x240 rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/image: load webcam-url show face
            ]
        ]
    ]
    webcam2: image load webcam-url 320x240 rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/image: load webcam-url show face
            ]
        ]
    ]
]
```

Unfortunately, this technique is not asynchronous. Each piece of event code is actually executed consecutively, in an alternating pattern, instead of simultaneously. Although the effect is similar (even indistinguishable) in many cases, the evaluation of code is not concurrent. For example, the following example adds a time display to the webcam viewer. You'll see that the clock is not updated every second. That's because the image download code and the clock code run alternately. The image download must be completed *before* the clock's 'time action can be evaluated. Try stopping the video to see the difference:

```
webcam-url: http://209.165.153.2/axis-cgi/jpg/image.cgi
view layout [
    btn "Start Video" [
        webcam/rate: 0
        webcam/image: load webcam-url
        show webcam
    ]
    btn "Stop Video" [webcam/rate: none show webcam]
    return
    webcam: image load webcam-url 320x240 rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/image: load webcam-url show face
            ]
        ]
    ]
    clock: field to-string now/time/precise rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/text: to-string now/time/precise show face
            ]
        ]
    ]
]
```

One solution to achieving truly asynchronous activity is to simply write the code for one process into a separate file and run it in a separate REBOL interpreter process using the "launch" function:

```
write %async.r {
    REBOL []
```

```
    view layout [
        clock: field to-string now/time/precise rate 0 feel [
            engage: func [face action event][
                if action = 'time [
                    face/text: to-string now/time/precise show face
                ]
            ]
        ]
    ]
}

launch %async.r
; REBOL will NOT wait for the evaluation of code in async.r
; to complete before going on:

webcam-url: http://209.165.153.2/axis-cgi/jpg/image.cgi
view layout [
    btn "Start Video" [
        webcam/rate: 0
        webcam/image: load webcam-url
        show webcam
    ]
    btn "Stop Video" [webcam/rate: none show webcam]
    return
    webcam: image load webcam-url 320x240 rate 0 feel [
        engage: func [face action event][
            if action = 'time [
                face/image: load webcam-url show face
            ]
        ]
    ]
]
```

The technique above simply creates two totally separate REBOL programs from within a single code file. If such programs need to interact, share data, or respond to interactive activity states, they can communicate via tcp network port, or by reading/writing data via a shared storage device.

9.8 Using DLLs and Shared Code Files in REBOL

"Dll"s in Windows, "So" files in Linux, and "Dylib" on Macs are *libraries of functions* that can be shared among different programming languages. Shared code libraries are used to *extend the capabilities of a language with* ***new functions***. They allow you to accomplish goals which aren't possible (or which are otherwise complicated) using the native functions built into the language. Most of the executable code, *and all the potential capabilities*, of most operating systems is contained in such files. Third party code libraries are also available to make easy work of complex tasks such as multimedia programming, 3d game programming, specialized hardware control, etc. To use Dlls and shared code files in REBOL, you'll need to download version 2.76 or later of the REBOL interpreter (rebview.exe). If you're using any of the beta versions from http://www.rebol.net/builds/, use either rebview.exe or rebcmdview.exe to run the examples in this section.

Using the format below, you can access and use the functions contained in most DLLs, *as if they're native REBOL functions*:

```
lib: load/library %TheNameOfYour.DLL

; "TheFunctionNameInsideTheDll" is loaded from the Dll and converted
; into a new REBOL function called "your-rebol-function-name":

your-rebol-function-name: make routine! [
    return-value: [data-type!]
    first-parameter [data-type!]
    another-parameter [data-type!]
    more-parameters [and-their-data-types!]
    ...
] lib "TheFunctionNameInsideTheDll"

; When the new REBOL function is used, it actually runs the function
; inside the Dll:

your-rebol-function-name parameter1 parameter2 ...

free lib
```

The first line opens access to the functions contained in the specified Dll. The following lines convert the function contained in the Dll to a format that can be used in REBOL. To make the conversion, a REBOL function is labeled and defined (i.e, "your-rebol-function-name" above), and a block containing the labels and types of parameters used and values returned from the function must be provided ("[return: [integer!]]" and "first-parameter [data-type!] another-parameter [data-type!] more-parameters [and-their-data-types!]" above). The name of the function, as labeled in the Dll, must also be provided immediately after the parameter block ("TheFunctionNameInsideTheDll" above). The second to last line above actually executes the new REBOL function, using any appropriate parameters you choose. When you're done using functions from the Dll, the last line is used to free up the Dll so that it's closed by the operating system.

Here are some examples:

```
REBOL []

; The "kernel32.dll" is a standard dll in all Windows installations:

lib: load/library %kernel32.dll

; The "beep" function is contained in the kernel32.dll library.
; We'll create a new REBOL function called "play-sound" that
; actually executes the "beep" function in kernel32.dll.  The
```

```
; "beep" function takes two integer parameters (pitch and
; duration values), and returns an integer value:

play-sound: make routine! [
    return: [integer!] pitch [integer!] duration [integer!]
] lib "Beep"

; (Beep returns a value of zero if the function does not complete
; successfully.  Otherwise it returns a nonzero number).

; Now we can use the "play-sound" function AS IF IT'S A NATIVE
; REBOL FUNCTION:

for hertz 37 3987 50 [
    print rejoin ["The pitch is now " hertz " hertz."]
    play-sound hertz 50
]

free lib
halt
```

The following example demonstrates how to record sounds (with the microphone attached to your computer) using the Windows MCI API. When complete, the recorded sound is played back using a native REBOL sound port:

```
; Various mci functions are included in the winmm.dll library.
; We'll create a new REBOL function called "mciExecute" that
; allows us to run MCI functions in winmm.dll.  This function
; function takes one string parameter (a text string written
; in MCI function syntax), and returns an integer value (true
; if the function is successful, false if it fails):

lib: load/library %winmm.dll
mciExecute: make routine! [
    command [string!]
    return: [logic!]
] lib "mciExecute"

; Get a file name from the user, which will be used to save the
; recorded sound:

file: to-local-file to-file request-file/save/title/file "Save as:" {
    } %rebol-recording.wav

; Open an MCI buffer and begin the recording:

mciExecute "open new type waveaudio alias buffer1 buffer 6"
mciExecute "record buffer1"

ask "RECORDING STARTED (press [ENTER] when done)...^/"

; Stop recording and save the sound to the wave file selected above:

mciExecute "stop buffer1"
mciExecute join "save buffer1 " file

; Close the DLL:

free lib

print "Recording complete.  Here's how it sounds:^/"
```

```
; Play back the sound:

insert port: open sound:// load to-rebol-file file wait port close port
print "DONE.^/"

halt
```

The next example demonstrates how to play AVI video files, again using the Windows API "mciExecute" from winmm.dll. A demo video is downloaded from the Internet and played two times - once with default settings, and a second time at a given location on screen at twice the original recorded speed. The video codec in the demo video is MS-CRAM (Microsoft Video 1), and the audio format is PCM. For more information about mciExecute commands, Google "multimedia command strings" and see http://msdn.microsoft.com/en-us/library/dd743572(VS.85).aspx:

```
; These lines open the winmm.dll and define the "mciExecute" function
; in REBOL:

lib: load/library %winmm.dll
mciExecute: make routine! [c [string!] return: [logic!]] lib "mciExecute"

; These lines download a demo video:

if not exists? %test.avi [
    flash "Downloading test video..."
    write/binary %test.avi read/binary http://re-bol.com/test.avi
    unview
]
video: to-local-file %test.avi

; The lines run the mciExecute function with the commands needed to
; play the video:

mciExecute rejoin ["OPEN " video " TYPE AVIVIDEO ALIAS thevideo"]
mciExecute "PLAY thevideo WAIT"
mciExecute "CLOSE thevideo"

mciExecute rejoin ["OPEN " video " TYPE AVIVIDEO ALIAS thevideo"]
mciExecute "PUT thevideo WINDOW AT 200 200 0 0"  ; at 200x200
mciExecute "SET thevideo SPEED 2000"  ; play twice a fast
mciExecute "PLAY thevideo WAIT"
mciExecute "CLOSE thevideo"

; These lines clean up:

free lib
quit
```

The next example uses the "dictionary.dll" from http://www.reelmedia.org/pureproject/archive411/dll/Dictionary.zip to perform a spell check on text entered at the REBOL command line. There are two functions in the dll that are required to perform a spell check - "Dictionary_Load" and "Dictionary_Check":

```
REBOL []

check-me: ask "Enter a word to be spell-checked:  "

lib: load/library %Dictionary.dll
```

```
; Two new REBOL functions are created, which actually run the
; Dictionary_Load and Dictionary_Check functions in the DLL:

load-dic: make routine! [
    a [string!]
    return: [none]
] lib "Dictionary_Load"

check-word: make routine! [
    a [string!]
    b [integer!]
    return: [integer!]
] lib "Dictionary_Check"

; This line runs the Dictionary_Load function from the DLL:

load-dic ""

; This line runs the Dictionary_Check function in the DLL, on
; whatever text was entered into the "check-me" variable above:

response: check-word check-me 0

; The Dictionary_Check function returns 0 if there are no errors:

either response = 0 [
    print "No spelling errors found."
] [
    print "That word is not in the dictionary."
]

free lib
halt
```

The following example plays an mp3 sound file using the Dll at http://musiclessonz.com/mp3.dll. Of course, that Dll could be compressed and embedded in the code to eliminate the necessity of downloading the file:

```
REBOL []

write/binary %mp3.dll read/binary http://musiclessonz.com/mp3.dll
lib: load/library %mp3.dll

; the "playfile" function is loaded from the Dll, and converted
; to a new REBOL "play-mp3" function:

play-mp3: make routine! [a [string!] return: [none]] lib "playfile"

; Then an mp3 file name is requested from the user, which is played
; by the "playfile" function in the Dll:

file: to-local-file to-string request-file
play-mp3 file

print "Done playing, Press [Esc] to quit this program: "
free lib
```

The next example uses the "AU3_MouseMove" function from the Dll version of AutoIt, to move the mouse around the screen. AutoIt contains a wide variety of functions to programatically push buttons, type text, select menu items, choose items from lists, control the mouse, etc. in any existing program window, as if those actions had been performed by a user clicking and typing on screen. Learning

the other functions in the AutoIt language can be very helpful in customizing and automating existing Windows applications:

```
REBOL []

if not exists? %AutoItDLL.dll [
    write/binary %AutoItDLL.dll
    read/binary http://musiclessonz.com/rebol_tutorial/AutoItDLL.dll
]

lib: load/library %AutoItDLL.dll
move-mouse: make routine! [
    return: [integer!] x [integer!] y [integer!] z [integer!]
] lib "AUTOIT_MouseMove"

print "Press the [Enter] key to see your mouse move around the screen."
print "It will move to the top corner, and then down diagonally to"
ask "position 200x200:  "

for position 0 200 5 [
    move-mouse position position 10
    ; "10" refers to the speed of the mouse movement
]

free lib
print "^/Done.^/"
halt
```

This example uses Dll functions from the native Windows API to adjust the title bar in your REBOL programs. Just include this code in your script if you need to eliminate the default 'REBOL - ' text at the top of your GUI programs:

```
REBOL []

; First, load the necessary dll:

user32.dll: load/library %user32.dll

; Then define the Windows API functions you'll need:

get-focus: make routine! [return: [int]] user32.dll "GetFocus"

set-caption: make routine! [
    hwnd [int]
    a [string!]
    return: [int]
] user32.dll "SetWindowTextA"

; Next, create your GUI - be sure to use 'view/new', so that it doesn't
; appear immediately (start the GUI later with 'do-events', after you've
; changed the title bar below):

view/new center-face layout [
    backcolor white
    text bold "Notice that there's no 'Rebol - ' in the title bar above."
    text "New title text:"
    across
    f: field "Tada!"
    btn "Change Title" [
        ; These functions change the text in the title bar:
```

```
        hwnd: get-focus
        set-caption hwnd f/text
    ]
    btn "Exit" [
        ; Be sure to close the dll when you're done:
        free user32.dll
        quit
    ]
]

; Once you've created your GUI, run the Dll functions to replace the
; default text in the title bar:

hwnd: get-focus
set-caption hwnd "My Title"

; Finally, start your GUI:

do-events
```

The following application demonstrates how to use the Windows API to view video from a local web cam, to save snapshots in BMP format, and to change the REBOL GUI window title:

```
REBOL []

; First, open the Dlls that contain the Windows API functions we want
; to use (to view webcam video, and to change window titles):

avicap32.dll: load/library %avicap32.dll
user32.dll: load/library %user32.dll

; Create REBOL function prototypes required to change window titles:
; (These functions are found in user32.dll, built in to Windows.)

get-focus: make routine! [return: [int]] user32.dll "GetFocus"
set-caption: make routine! [
    hwnd [int] a [string!]  return: [int]
] user32.dll "SetWindowTextA"

; Create REBOL function prototypes required to view the webcam:
; (also built in to Windows)

find-window-by-class: make routine! [
    ClassName [string!] WindowName [integer!] return: [integer!]
] user32.dll "FindWindowA"
sendmessage: make routine! [
    hWnd [integer!] val1 [integer!] val2 [integer!] val3 [integer!]
    return: [integer!]
] user32.dll "SendMessageA"
sendmessage-file: make routine! [
    hWnd [integer!] val1 [integer!] val2 [integer!] val3 [string!]
    return: [integer!]
] user32.dll  "SendMessageA"
cap: make routine! [
    cap [string!] child-val1 [integer!] val2 [integer!] val3 [integer!]
    width [integer!] height [integer!] handle [integer!]
    val4 [integer!] return: [integer!]
] avicap32.dll "capCreateCaptureWindowA"

; Create the REBOL GUI window:
```

```
view/new center-face layout/tight [
    image 320x240
    across
    btn "Take Snapshot" [
        ; Run the dll functions that take a snapshot:
        sendmessage cap-result 1085 0 0
        sendmessage-file cap-result 1049 0 "scrshot.bmp"
    ]
    btn "Exit" [
        ; Run the dll functions that stop the video:
        sendmessage cap-result 1205 0 0
        sendmessage cap-result 1035 0 0
        free user32.dll
        quit
    ]
]

; Run the Dll functions that reset our REBOL GUI window title:
; (eliminates "REBOL - " in the title bar)

hwnd-set-title: get-focus
set-caption hwnd-set-title "Web Camera"

; Run the Dll functions that show the video:

hwnd: find-window-by-class "REBOLWind" 0
cap-result: cap "cap" 1342177280 0 0 320 240 hwnd 0
sendmessage cap-result 1034 0 0
sendmessage cap-result 1077 1 0
sendmessage cap-result 1075 1 0
sendmessage cap-result 1074 1 0
sendmessage cap-result 1076 1 0

; start the GUI:

do-events
```

For more information about DLLs and the Windows API, see:

http://rebol.com/docs/library.html
http://en.wikipedia.org/wiki/Dynamic_Link_Library
http://www.math.grin.edu/~shirema1/docs/DLLsinREBOL.html
http://www.borland.com/devsupport/borlandcpp/patches/BC52HLP1.ZIP
http://www.allapi.net/downloads/apiguide/agsetup.exe
http://www.activevb.de/rubriken/apiviewer/index-apiviewereng.html
http://msdn.microsoft.com/library/

Remember, whenever you use any Dll or code created by another programmer, be absolutely sure to check, and follow, the licensing terms by which it's distributed.

9.9 Web Programming and the CGI Interface

In "CGI" web applications, HTML forms on a web site act as the user interface (GUI) for scripts that run on a web server. Users typically type text into fields, select choices from drop down lists, click check boxes, and otherwise enter data into form widgets on a web page, and then click a "submit" button when done. The submitted data is transferred to, and processed by, a script that you've stored at a specified URL (Internet address) on your web server. Data output from the script is then sent back to the user's browser and displayed on screen as a dynamically created web page. CGI programs of that sort, running on web sites, are among the most common types of computer application in contemporary use. PHP, Python, Java, PERL, and ASP are popular languages used to accomplish similar Internet programming tasks, but if you know REBOL, you don't need to learn them. REBOL's CGI interface makes Internet programming very easy.

In order to create REBOL CGI programs, you need an available web server. A web server is a computer attached to the Internet, which constantly runs a program that stores and sends out web page text and data, when requested from an Internet browser running on another computer. The most popular web serving application is Apache. Most small web sites are typically run on shared web server hosting accounts, rented from a data center for a few dollars per month (see http://www.lunarpages.com - they're REBOL friendly). While setting up a web server account, you can register an available *domain name* (i.e, www.yourwebsitename.com). When web site visitors type your ".com" domain address into their browser, they see files that you've created and saved into a publicly accessible file folder on your web server computer.

In order for REBOL CGI scripts to run, the REBOL interpreter must be installed on your web server. To do that, download from rebol.com the correct version of the REBOL interpreter for the operating system on which your web server runs (most often some type of Linux). Upload it to your user path on your web server, and **set the permissions to allow it to be executed** (typically "755"). Ask your web site host if you don't understand what that means. http://rebol.com/docs/cgi1.html#section-2.2 has some basic information about how to install REBOL on your server. If you don't have an online web server account, you can download a full featured free Apache web server package that will run on your local Windows PC, from http://www.uniformserver.com.

9.9.1 HTML

In order to create any sort of CGI application, you need to understand a bit about HTML. HTML is the layout language used to format text and GUI elements on all web pages. HTML is not a programming language - it doesn't have facilities to process or manipulate data. It's simply a markup format that allows you to shape the visual appearance of text, images, and other items on pages viewed in a browser.

In HTML, items on a web page are enclosed between starting and ending "tags":

```
<STARTING TAG>Some item to be included on a web page</ENDING TAG>
```

There are tags to effect the layout in every possible way. To bold some text, for example, surround it in opening and closing "strong" tags:

```
<STRONG>some bolded text</STRONG>
```

The code above appears on a web page as: **some bolded text**.

To italicize text, surround it in < i > and < / i > tags:

```
<i>some italicized text</i>
```

That appears on a web page as: *some italicized text*.

To create a table with three rows of data, do the following:

```
<TABLE border=1>
    <TR><TD>First Row</TD></TR>
    <TR><TD>Second Row</TD></TR>
    <TR><TD>Third Row</TD></TR>
</TABLE>
```

Notice that every

```
<opening tag>
```

in HTML code is followed by a corresponding

```
</closing tag>
```

Some tags surround all of the page, some tags surround portions of the page, and they're often nested inside one another to create more complex designs.

A minimal format to create a web page is shown below. Notice that the title is nested between "head" tags, and the entire document is nested within "HTML" tags. The page content seen by the user is surrounded by "body" tags:

```
<HTML>
    <HEAD>
        <TITLE>Page title</TITLE>
    </HEAD>
    <BODY>
        A bunch of text and <i>HTML formatting</i> goes here...
    </BODY>
</HTML>
```

If you save the above code to a text file called "yourpage.html", upload it to your web server, and surf to http://yourwebserver.com/yourpage.html , you'll see in your browser a page entitled "Page title", with the text "A bunch of text and *HTML formatting* goes here...". All web pages work that way - this tutorial is in fact just an HTML document stored on the author's web server account. Click View -> Source in your browser, and you'll see the HTML tags that were used to format this document.

9.9.2 HTML Forms and Server Scripts - the Basic CGI Model

The following HTML example contains a "form" tag inside the standard HTML head and body layout. Inside the form tags are a text input field tag, and a submit button tag:

```
<HTML>
    <HEAD><TITLE>Data Entry Form</TITLE></HEAD>
    <BODY>
        <FORM ACTION="http://yourwebserver.com/your_rebol_script.cgi">
            <INPUT TYPE="TEXT" NAME="username" SIZE="25">
            <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM>
    </BODY>
</HTML>
```

Forms can contain tags for a variety of input types: multi-line text areas, drop down selection boxes, check boxes, etc. See http://www.w3schools.com/html/html_forms.asp for more information about

form tags.

You can use the data entered into any form by pointing the *action* address to the URL at which a specific REBOL script is located. For example, 'FORM ACTION="http://yourwebserver.com /your_rebol_script.cgi"' in the above form could point to the URL of the following CGI script, which is saved as a text file on your web server. When a web site visitor clicks the submit button in the above form, the data is sent to the following program, which in turn does some processing, and *prints output directly to the user's web browser*. NOTE: Remember that in REBOL *curly brackets are the same as quotes*. Curly brackets are used in all the following examples, because they allow for multiline content and they help improve readability by clearly showing where strings begin and end:

```
#!/home/your_user_path/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}
; the line above is the same as:  print "content-type: text/html^/"
submitted: decode-cgi system/options/cgi/query-string

print {<HTML><HEAD><TITLE>Page title</TITLE></HEAD><BODY>}
print rejoin [{Hello } second submitted {!}]
print {</BODY></HTML>}
```

In order for the above code to actually run on your web server, a working REBOL interpreter must be installed in the path designated by "/home/your_user_path/rebol/rebol -cs".

The first 4 lines of the above script are basically stock code. Include them at the top of every REBOL CGI script. Notice the *"decode-cgi"* line - it's the key to retrieving data submitted by HTML forms. In the code above, the decoded data is assigned the variable name "submitted". The submitted form data can be manipulated however desired, and *output is then returned to the user via the "print" function*. That's important to understand: *all data "print"ed by a REBOL CGI program appears directly in the user's web browser* (i.e., to the web visitor who entered data into the HTML form). The printed data is typically laid out with HTML formatting, so that it appears as a nicely formed web page in the user's browser.

Any normal REBOL code can be included in a CGI script - you can perform any type of data storage, retrieval, organization, and manipulation that can occur in any other REBOL program. The CGI interface just allows your REBOL code to run online on your web server, and for data to be input/output via web pages which are also stored on the web server, accessible by any visitor's browser.

9.9.3 A Standard CGI Template to Memorize

Most short CGI programs typically *print an initial HTML form to obtain data from the user*. In the initial printed form, the *action address typically points back to the **same** URL address as the script itself*. The script examines the submitted data, and if it's empty (i.e., no data has been submitted), the program prints the initial HTML form. Otherwise, it manipulates the submitted data in way(s) you choose and then prints some output to the user's web browser in the form of a new HTML page. Here's a basic example of that process, using the code above:

```
#!/home/your_user_path/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}
submitted: decode-cgi system/options/cgi/query-string

; The 4 lines above are the standard REBOL CGI headers.
; The line below prints the standard HTML, head and body
; tags to the visitor's browser:

print {<HTML><HEAD><TITLE>Page title</TITLE></HEAD><BODY>}
```

```
; Next, determine if any data has been submitted.
; Print the initial form if empty.  Otherwise, process
; and print out some HTML using the submitted data.
; Finally, print the standard closing "body" and "html"
; tags, which were opened above:

either empty? submitted [
    print {
        <FORM ACTION="http://yourwebserver.com/this_rebol_script.cgi">
        <INPUT TYPE="TEXT" NAME="username" SIZE="25">
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM>
        </BODY></HTML>
    }
] [
    print rejoin [{Hello } second submitted {!}]
    print {</BODY></HTML>}
]
```

Using the above standard outline, you can include any required HTML form(s), along with all executable code and data required to make a complete CGI program, all in one script file. *Memorize it*.

9.9.4 Examples

Here's a REBOL CGI form-mail program that prints an initial form, then sends an email to a given address containing the user-submitted data:

```
#!/home/youruserpath/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}
submitted: decode-cgi system/options/cgi/query-string

; the following account info is required to send email:

set-net [from_address@website.com  smtp.website.com]

; print a more complicated HTML header:

print read %template_header.html

; if some form data has been submitted to the script:

if not empty? submitted [
    sent-message: rejoin [
        newline {INFO SUBMITTED BY WEB FORM} newline newline
        {Time Stamp: } (now + 3:00) newline
        {Name: } submitted/2 newline
        {Email: } submitted/4 newline
        {Message: } submitted/6 newline
    ]

    send/subject to_address@website.com sent-message "FORM SUBMISSION"

    html: copy {}
    foreach [var value] submitted [
        repend html [<TR><TD> mold var </TD><TD> mold value </TD></TR>]
    ]
    print {<font size=5>Thank You!</font> <br><br>
        The following information has been sent: <BR><BR>}
    print rejoin [{Time Stamp: } now + 3:00]
```

```
    print {<BR><BR><table>}
    print html
    print {</table>}
    ;  print a more complicated HTML footer:
    print read %template_footer.html
    quit
]

; if no form data has been submitted, print the initial form:

print {
    <CENTER><TABLE><TR><TD>
    <BR><strong>Please enter your info below:</strong><BR><BR>
    <FORM ACTION="http://yourwebserver.com/this_rebol_script.cgi">
    Name: <BR> <INPUT TYPE="TEXT" NAME="name"><BR><BR>
    Email: <BR> <INPUT TYPE="TEXT" NAME="email"><BR><BR>
    Message: <BR>
    <TEXTAREA cols=75 name=message rows=5></TEXTAREA> <BR><BR>
    <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
    </FORM>
    </TD></TR></TABLE></CENTER>
}
print read %template_footer.html
```

The template_header.html file used in the above example can include the standard required HTML outline, along with any formatting tags and static content that you'd like, in order to present a nicely designed page. A basic layout may include something similar to the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Page Title</TITLE>
<META http-equiv=Content-Type content="text/html;
    charset=windows-1252">
</HEAD>
<BODY bgColor=#000000>
<TABLE align=center background="" border=0
    cellPadding=20 cellSpacing=2 height="100%" width="85%">
<TR>
<TD background="" bgColor=white vAlign=top>
```

The footer closes any tables or tags opened in the header, and may include any static content that appears after the CGI script (copyright info, logos, etc.):

```
</TD>
</TR>
</TABLE>
<TABLE align=center background="" border=0
    cellPadding=20 cellSpacing=2 width="95%">
<TR>
<TD background="" cellPadding=2 bgColor=#000000 height=5>
<P align=center><FONT color=white size=1>Copyright © 2009
    Yoursite.com.  All rights reserved.</FONT>
</P>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

The following example demonstrates how to automatically build lists of days, months, times, and

data read from a file, using dynamic loops (foreach, for, etc.). The items are selectable from drop down lists in the printed HTML form:

```
#!/home/youruserpath/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}
submitted: decode-cgi system/options/cgi/query-string

print {<HTML><HEAD><TITLE>Dropdown Lists</TITLE></HEAD><BODY>}

if not empty? submitted [
    print rejoin [{NAME SELECTED: } submitted/2 {<BR><BR>}]
    selected: rejoin [
        {TIME/DATE SELECTED: }
        submitted/4 { } submitted/6 {, } submitted/8
    ]
    print selected
    quit
]

; If no data has been submitted, print the initial form:

print {<FORM ACTION="http://yourwebserver.com/your_rebol_script.cgi">
   SELECT A NAME:  <BR> <BR>}
names: read/lines %users.txt
print {<select NAME="names">}
foreach name names [prin rejoin [{<option>} name]]
print {</option> </select> <br> <br>}

print { SELECT A DATE AND TIME: }
print rejoin [{(today's date is } now/date {)} <BR><BR>]

print {<select NAME="month">}
foreach m system/locale/months [prin rejoin [{<option>} m]]
print {</option> </select>}

print {<select NAME="date">}
for daysinmonth 1 31 1 [prin rejoin [{<option>} daysinmonth]]
print {</option> </select>}

print {<select NAME="time">
for time 10:00am 12:30pm :30 [prin rejoin [{<option>} time]]
for time 1:00 10:00 :30 [prin rejoin [{<option>} time]]
print {</option> </select> <br> <br>}

print {<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit"></FORM>}
```

The "users.txt" file used in the above example may look something like this:

```
nick
john
jim
bob
```

Here's a simple CGI program that displays all photos in the current folder on a web site, using a foreach loop:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "Photo Viewer"]
```

```
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>Photos</TITLE></HEAD><BODY>}
print read %template_header.html

folder: read %.
count: 0
foreach file folder [
    foreach ext [".jpg" ".gif" ".png" ".bmp"] [
        if find file ext [
            print [<BR> <CENTER>]
            print rejoin [{<img src="} file {">}]
            print [</CENTER>]
            count: count + 1
        ]
    ]
]
print {<BR>}
print rejoin [{Total Images: } count]
print read %template_footer.html
```

Notice that there's no "submitted: decode-cgi system/options/cgi/query-string" code in the above example. That's because the above script doesn't make use of any data submitted from a form.

Here's a simple but powerful script that allows you to type REBOL code into an HTML text area, and have that code execute directly on your web server. The results of the code are then displayed in your browser. This essentially functions as a remote console for the REBOL interpreter on your server. You can use it to run REBOL code, or to call shell programs directly on your web site - very powerful! *DO NOT run this on your web server if you're concerned at all about security!*:

```
#! /home/path/public_html/rebol/rebol276 -cs
REBOL [Title: "CGI Remote Console"]
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>Console</TITLE></HEAD><BODY>}
submitted: decode-cgi system/options/cgi/query-string

; If no data has been submitted, print form to request user/pass:

if ((submitted/2 = none) or (submitted/4 = none)) [
    print {
        <STRONG>W A R N I N G  -  Private Server, Login Required:</STRONG>
        <BR><BR>
        <FORM ACTION="./console.cgi">
        Username: <INPUT TYPE=text SIZE="50" NAME="name"><BR><BR>
        Password: <INPUT TYPE=text SIZE="50" NAME="pass"><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM>
    }
    quit
]

; If code has been submitted, print the output:

entry-form: [
    print {
        <CENTER><FORM METHOD="get" ACTION="./console.cgi">
        <INPUT TYPE=hidden NAME=submit_confirm VALUE="command-submitted">
        <TEXTAREA COLS="100" ROWS="10" NAME="contents"></TEXTAREA><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM></CENTER></BODY></HTML>
    }
]
```

```
if submitted/2 = "command-submitted" [
    write %commands.txt join "REBOL[]^/" submitted/4
    ; The "call" function requires REBOL version 2.76:
    call/output/error
        {/home/path/public_html/rebol/rebol276 -qs commands.txt}
        %conso.txt %conse.txt
    do entry-form
    print rejoin [
        {<CENTER>Output: <BR><BR>}
        {<TABLE WIDTH=80% BORDER="1" CELLPADDING="10"><TR><TD><PRE>}
        read %conso.txt
        {</PRE></TD></TR></TABLE><BR><BR>}
        {Errors: <BR><BR>}
        read %conse.txt
        {</CENTER>}
    ]
    quit
]

; Otherwise, check submitted user/pass, then print form for code entry:

username: submitted/2 password: submitted/4
either (username = "user") and (password = "pass") [
    ; if user/pass is ok, go on
][
    print "Incorrect Username/Password." quit
]

do entry-form
```

Upload the script to your server, rename it "console.cgi", set it to executable, and change the path to your REBOL interpreter (2 places in the script). Then try running the following example code:

```
print 352 + 836
? system/locale/months
call "ls -al"
```

Here's an example that allows users to check attendance at various weekly events, and add/remove their names from each of the events. It stores all the user information in a flat file (simple text file) named "jams.db":

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "event.cgi"]
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>Event Sign-Up</TITLE></HEAD><BODY>}

jams: load %jam.db

a-line: [] loop 65 [append a-line "-"]
a-line: trim to-string a-line

print {
    <hr> <font size=5>" Sign up for an event:"</font> <hr><BR>
    <FORM ACTION="http://yourwebsite.com/cgi-bin/event.cgi">
    Student Name:
    <input type=text size="50" name="student"><BR><BR>
    ADD yourself to this event:              "
    <select NAME="add"><option>""<option>"all"
```

```
}
foreach jam jams [prin rejoin [{<option>} jam/1]]
print {
    </option> </select> <BR> <BR>
    REMOVE yourself from this event:
    <select NAME="remove"><option>""<option>"all"
}
foreach jam jams [prin rejoin [{<option>} jam/1]]
print {
    </option> </select> <BR> <BR>
    <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
    </FORM>
}

print-all: does [
    print [<br><hr><font size=5>]
    print " Currently scheduled events, and current attendance:"
    print [</font><br>]
    foreach jam jams [
        print rejoin [a-line {<BR>} jam/1 {BR} a-line {<BR>}]
        for person 2 (length? jam) 1 [
            print jam/:person
            print {<BR>}
        ]
        print {<BR>}
    ]
    print {</BODY></HTML>}
]

submitted: decode-cgi system/options/cgi/query-string

if submitted/2 <> none [
    if ((submitted/4 = "") and (submitted/6 = "")) [
        print {
            <strong> Please try again. You must choose an event.</strong>
        }
        print-all
        quit
    ]
    if ((submitted/4 <> "") and (submitted/6 <> "")) [
        print {
            <strong> Please try again. Choose add OR remove.</strong>
        }
        print-all
        quit
    ]
    if submitted/4 = "all" [
        foreach jam jams [append jam submitted/2]
        save %jam.db jams
        print {
            <strong> Your name has been added to every event:</strong>
        }
        print-all
        quit
    ]
    if submitted/6 = "all" [
        foreach jam jams [
            if find jam submitted/2 [
                remove-each name jam [name = submitted/2]
                save %jam.db jams
            ]
        ]
        print {
```

```
            <strong> Your name has been removed from all events:</strong>
        }
        print-all
        quit
    ]
    foreach jam jams [
        if (find jam submitted/4) [
            append jam submitted/2
            save %jam.db jams
            print {
                <strong> Your name has been added to the selected event:
                </strong>
            }
            print-all
            quit
        ]
    ]
    found: false
    foreach jam jams [
        if (find jam submitted/6) [
            if (find jam submitted/2) [
                remove-each name jam [name = submitted/2]
                save %jam.db jams
                print {
                    <strong>
                    Your name has been removed from the selected event:
                    </strong>
                }
                print-all
                quit
                found: true
            ]
        ]
    ]
    if found <> true [
        print {
            <strong> That name is not found in the specified event!"
            </strong>
        }
        print-all
        quit
    ]
]

print-all
```

Here is a sample of the "jam.db" data file used in the above example:

```
["Sunday September 16, 4:00 pm - Jam CLASS"
    "Nick Antonaccio" "Ryan Gaughan" "Mark Carson"]
["Sunday September 23, 4:00 pm - Jam CLASS"
    "Nick Antonaccio" "Ryan Gaughan" "Mark Carson"]
["Sunday September 30, 4:00 pm - Jam CLASS"
    "Nick Antonaccio" "Ryan Gaughan" "Mark Carson"]
```

Here's a simple web site bulletin board program:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "Jam"]
print {content-type: text/html^/}
```

```
print read %template_header.html
; print {<HTML><HEAD><TITLE>Bulletin Board</TITLE></HEAD><BODY>}

bbs: load %bb.db

print {
    <center><table border=1 cellpadding=10 width=600><tr><td>
    <center><strong><font size=4>
    Please REFRESH this page to see new messages.
    </font></strong></center>
}

print-all: does [
    print {<br><hr><font size=5> Posted Messages: </font> <br><hr>}
    foreach bb (reverse bbs) [
        print rejoin [
            {<BR>Date/Time: } bb/2 {          }
            {"Name: } bb/1 {<BR><BR>} bb/3 {<BR><BR><HR>}
        ]
    ]
]

submitted: decode-cgi system/options/cgi/query-string

if submitted/2 <> none [
    entry: copy []
    append entry submitted/2
    append entry to-string (now + 3:00)
    append entry submitted/4
    append/only bbs entry
    save %bb.db bbs
    print {<BR><strong>Your message has been added: </strong><BR>}
]

print-all

print {
    <font size=5> Post A New Public Message:</font><hr>
    <FORM ACTION="http://website.com/bb/bb.cgi">
    <br> Your Name:  <br>
    <input type=text size="50" name="student"><BR><BR>
    Your Message: <br>
    <textarea name=message rows=5 cols=50></textarea><BR><BR>
    <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Post Message">
    </FORM>
    </td></tr></table></center>
}
print read %template_footer.html
```

Here's an example data file for the program above:

```
[
    [
        "Nick Antonaccio"
        "8-Nov-2006/4:55:59-8:00"
        {
            WELCOME TO OUR PUBLIC BULLETIN BOARD.
             Please keep the posts clean cut and on topic.
            Thanks and have fun!
        }
    ]
```

```
]
```

The default format for REBOL CGI data is "GET". Data submitted by the GET method in an HTML form is displayed in the URL bar of the user's browser. If you don't want users to see that data displayed, or if the amount of submitted data is larger then can be contained in the URL bar of a browser, the "POST" method should be used. To work with the POST method, the action in your HTML form should be:

```
<FORM METHOD="post" ACTION="./your_script.cgi">
```

You must also use the "read-cgi" function below to decode the submitted POST data in your REBOL script. This example creates a password protected online text editor, with an automatic backup feature:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>Edit Text Document</TITLE></HEAD><BODY>}

; submitted: decode-cgi system/options/cgi/query-string

; We can't use the normal line above to decode, because
; we're using the POST method to submit data (because data
; from the textarea may get too big for the GET method).
; Use the following standard function to process data from
; a POST method instead:

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: decode-cgi read-cgi

; if document.txt has been edited and submitted:

if submitted/2 = "save" [
    ; save newly edited document:
    write to-file rejoin ["./" submitted/6 "/document.txt"] submitted/4
    print ["Document Saved."]
    print rejoin [
        {<META HTTP-EQUIV="REFRESH" CONTENT="0;
            URL=http://website.com/folder/}
        submitted/6 {">}
    ]
    quit
]

; if user is just opening page (i.e., no data has been submitted
```

```
; yet), request user/pass:

if ((submitted/2 = none) or (submitted/4 = none)) [
    print {
        <strong>W A R N I N G  -  Private Server, Login Required:
        </strong><BR><BR>
        <FORM ACTION="./edit.cgi">
        Username: <input type=text size="50" name="name"><BR><BR>
        Password: <input type=text size="50" name="pass"><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM>
    }
    quit
]

; check user/pass against those in userlist.txt,
; end program if incorrect:

userlist: load %userlist.txt
folder: submitted/2
password: submitted/4
response: false
foreach user userlist [
    if ((first user) = folder) and (password = (second user)) [
        response: true
    ]
]
if response = false [print {Incorrect Username/Password.} quit]

; if user/pass is ok, go on...

; backup (before changes are made):

cur-time: to-string replace/all to-string now/time {:} {-}
document_text: read to-file rejoin [{./} folder {/document.txt}]
write to-file rejoin [
    {./} folder {/} now/date {_} cur-time {.txt}] document_text

; note the POST method in the HTML form:

prin {
    <strong>Be sure to SUBMIT when done:</strong><BR><BR>
    <FORM method="post" ACTION="./edit.cgi">
    <INPUT TYPE=hidden NAME=submit_confirm VALUE="save">
    <textarea cols="100" rows="15" name="contents">
}
;
; The following line is what we want to do, but it won't work for
; HTML documents which contain <textarea>s
;
; print document_text
;
; The following line fixes the problem:
;
prin replace/all document_text {</textarea>} {&lt\/textarea&gt;}
print {</textarea><BR><BR>}
print rejoin [{<INPUT TYPE=hidden NAME=folder VALUE="} folder {">}]
print {<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">}
print {</FORM>}
print {</BODY></HTML>}
```

The following is a generic form handler that can be used to save GET or POST data to a text file. It's a useful replacement for generic form mailers, and makes the data much more accessible later by

other scripts:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print {content-type: text/html^/}

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]
submitted: decode-cgi read-cgi

print {
    <HTML><HEAD><TITLE>Your Form Has Been Submitted</TITLE></HEAD>
    <BODY><CENTER><TABLE border=0 cellPadding=10 width="80%"><TR><TD>
}

entry: rejoin [{[^/ "Time Stamp:" } {"} form (now + 3:00) {"^/}]
foreach [title value] submitted [
    entry: rejoin [entry { } {"} mold title {" } mold value {^/}]
]
append entry {]^/}
write/append %submitted_forms.txt entry

html: copy ""
foreach [title value] submitted [
    repend html [
        <TR><TD width=20%> mold title </TD><TD> mold value </TD></TR>
    ]
]

print rejoin [
    {
         <FONT size=5>Thank You! The following information has been
         submitted: </FONT><BR><BR>Time Stamp:
    }
    now + 3:00  {<BR><BR><TABLE border=1 cellPadding=10 width="100%">}
    html  {</TABLE><BR>}
    {
        To correct any errors or to submit forms for additional people,
        please click the [BACK] button in your browser, make any
        changes, and resubmit the form.  You'll hear from us shortly.
        Thank you!<BR><BR><CENTER><FONT size=1>
        Copyright © 2009 This Web Site.  All rights reserved.</FONT>
        </CENTER></TD></TR></CENTER></BODY></HTML>
    }
]

quit
```

The script below can be used on a desktop PC to easily view all the forms submitted at the script above. It provides nice GUI navigation, message count, sort by any data column, etc.:

```
REBOL [title: "CGI form submission viewer"]

sort-column: 4  ; even numered cols contain data (2nd col is time stamp)
signups: load http://yoursite.com/submitted_forms.txt
do create-list: [
    name-list: copy []
    foreach item signups [append name-list (pick item sort-column)]
]
view center-face layout [
    the-list: text-list 600x150 data name-list [
        foreach item signups [
            if (pick item sort-column) = value [
                dt: copy ""
                foreach [label data] item [
                    dt: rejoin [
                        dt newline label "  " data
                    ]
                ]
                a/text: dt
                show a
            ]
        ]
    ]
    a: area 600x300 across
    btn "Sort by..." [
        sort-column: to-integer request-text/title/default {
            Data column to list:} "4"
        do create-list
        the-list/data: name-list
        show the-list
    ]
    tab text join (form length? signups) " entries."
]
```

Here's another script that removes the title columns and reduces the form data into a usable format. Possibilities with managing form data like this are endless:

```
submissions: load http://yoursite.com/submitted_forms.txt
do create-list: [
    data: copy []
    foreach block submissions [append/only data (extract/index block 2 4)]
    datastring: copy {}
    foreach block data [
        datastring: join datastring "[^/"
        foreach item block [datastring: rejoin [datastring item newline]]
        datastring: join datastring "]^/^/"
    ]
    editor datastring
]
```

The following example demonstrates how to upload files to your web server using the decode-multipart-form-data function by Andreas Bolka:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [Title: "HTTP File Upload"]
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>File Upload</TITLE></HEAD>}
print {<BODY><br><br><center><table width=95% border=1>}
print {<tr><td width=100%><br><center>}
```

```
read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: read-cgi

if submitted/2 = none [
    print {
        <FORM ACTION="./upload.cgi"
        METHOD="post" ENCTYPE="multipart/form-data">
            <strong>Upload File:</strong><br><br>
            <INPUT TYPE="file" NAME="photo"> <br><br>
            <INPUT TYPE="submit" NAME="Submit" VALUE="Upload">
        </FORM>
        <br></center></td></tr></table></BODY></HTML>
    }
    quit
]

decode-multipart-form-data: func [
    p-content-type
    p-post-data
    /local list ct bd delim-beg delim-end non-cr non-lf non-crlf mime-part
] [
    list: copy []
    if not found? find p-content-type "multipart/form-data" [return list]
    ct: copy p-content-type
    bd: join "--" copy find/tail ct "boundary="
    delim-beg: join bd crlf
    delim-end: join crlf bd
    non-cr:     complement charset reduce [ cr ]
    non-lf:     complement charset reduce [ newline ]
    non-crlf:   [ non-cr | cr non-lf ]
    mime-part:  [
        ( ct-dispo: content: none ct-type: "text/plain" )
        delim-beg ; mime-part start delimiter
        "content-disposition: " copy ct-dispo any non-crlf crlf
        opt [ "content-type: " copy ct-type any non-crlf crlf ]
        crlf ; content delimiter
        copy content
        to delim-end crlf ; mime-part end delimiter
        ( handle-mime-part ct-dispo ct-type content )
    ]
    handle-mime-part: func [
        p-ct-dispo
        p-ct-type
        p-content
        /local tmp name value val-p
    ] [
        p-ct-dispo: parse p-ct-dispo {;="}
        name: to-set-word (select p-ct-dispo "name")
```

```
        either (none? tmp: select p-ct-dispo "filename")
               and (found? find p-ct-type "text/plain") [
            value: content
        ] [
            value: make object! [
                filename: copy tmp
                type: copy p-ct-type
                content: either none? p-content [none][copy p-content]
            ]
        ]
        either val-p: find list name
            [change/only next val-p compose [(first next val-p) (value)]]
            [append list compose [(to-set-word name) (value)]]
    ]
    use [ct-dispo ct-type content] [
        parse/all p-post-data [some mime-part "--" crlf]
    ]
    list
]

; After the following line, "probe cgi-object" will display all parts of
; the submitted multipart object:

cgi-object: construct decode-multipart-form-data
    system/options/cgi/content-type copy submitted

; Write file to server using the original filename, and notify the user:

the-file: last split-path to-file copy cgi-object/photo/filename
write/binary the-file cgi-object/photo/content
print {
    <strong>UPLOAD COMPLETE</strong><br><br>
    <strong>Files currently in this folder:</strong><br><br>
}
folder: sort read %.
foreach file folder [
    print [rejoin [{<a href="./} file {">} file {</a><br>}]]
]
print {<br></td></tr></table></BODY></HTML>}

; Alternatively, you could forward to a different page when done:
;
; wait 3
; refresh-me: {
;     <head><title></title>
;     <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.cgi"></head>
; }
; print refresh-me
```

This variation of the upload script allows you to select the directory to which files are uploaded:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [Title: "HTTP File Upload"]
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>File Upload</TITLE></HEAD>}
print {<BODY><br><br><center><table width=95% border=1>}
print {<tr><td width=100%><br><center>}

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
```

```
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: read-cgi

if submitted/2 = none [
    print {
        <FORM ACTION="./upload.cgi"
        METHOD="post" ENCTYPE="multipart/form-data">
            <strong>Upload File:</strong><br><br>
            <INPUT TYPE="file" NAME="photo"> <br><br>
            <INPUT TYPE="text" NAME="path" SIZE="35"
                VALUE="/home/path/public_html/">
            <INPUT TYPE="submit" NAME="Submit" VALUE="Upload">
        </FORM>
        <br></center></td></tr></table></BODY></HTML>
    }
    quit
]

decode-multipart-form-data: func [
    p-content-type
    p-post-data
    /local list ct bd delim-beg delim-end non-cr non-lf non-crlf mime-part
] [
    list: copy []
    if not found? find p-content-type "multipart/form-data" [return list]
    ct: copy p-content-type
    bd: join "--" copy find/tail ct "boundary="
    delim-beg: join bd crlf
    delim-end: join crlf bd
    non-cr:     complement charset reduce [ cr ]
    non-lf:     complement charset reduce [ newline ]
    non-crlf:   [ non-cr | cr non-lf ]
    mime-part:  [
        ( ct-dispo: content: none ct-type: "text/plain" )
        delim-beg ; mime-part start delimiter
        "content-disposition: " copy ct-dispo any non-crlf crlf
        opt [ "content-type: " copy ct-type any non-crlf crlf ]
        crlf ; content delimiter
        copy content
        to delim-end crlf ; mime-part end delimiter
        ( handle-mime-part ct-dispo ct-type content )
    ]
    handle-mime-part: func [
        p-ct-dispo
        p-ct-type
        p-content
        /local tmp name value val-p
    ] [
        p-ct-dispo: parse p-ct-dispo {;="}
        name: to-set-word (select p-ct-dispo "name")
        either (none? tmp: select p-ct-dispo "filename")
               and (found? find p-ct-type "text/plain") [
```

```
            value: content
        ] [
            value: make object! [
                filename: copy tmp
                type: copy p-ct-type
                content: either none? p-content [none][copy p-content]
            ]
        ]
        either val-p: find list name
            [change/only next val-p compose [(first next val-p) (value)]]
            [append list compose [(to-set-word name) (value)]]
    ]
    use [ct-dispo ct-type content] [
        parse/all p-post-data [some mime-part "--" crlf]
    ]
    list
]

cgi-object: construct decode-multipart-form-data
    system/options/cgi/content-type copy submitted

the-file: last split-path to-file copy cgi-object/photo/filename
write/binary the-file cgi-object/photo/content
print {
    <strong>UPLOAD COMPLETE</strong><br><br>
    <strong>Files currently in this folder:</strong><br><br>
}
folder: sort read to-file cgi-object/path
current-folder: rejoin  at
foreach file folder [
    print [rejoin [
        {<a href="http://site.com/"} (at cgi-object/path 28) file {">}
        ; convert path to URL
        file "</a><br>"
    ]]
]
print {<br></td></tr></table></BODY></HTML>}
```

Here's a script that demonstrates how to push download a file to the user's browser:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
root-path: "/home/path"

; if no data has been submitted, request file name:

if ((submitted/2 = none) or (submitted/4 = none)) [
    print "content-type: text/html^/"
    print [<STRONG>"W A R N I N G  -  "]
    print ["Private Server, Login Required:"</STRONG><BR><BR>]
    print [<FORM ACTION="./download.cgi">]
    print [" Username: " <INPUT TYPE=text SIZE="50" NAME="name"><BR><BR>]
    print [" Password: " <INPUT TYPE=text SIZE="50" NAME="pass"><BR><BR>]
    print [" File: "<BR><BR>]
    print [<INPUT TYPE=text SIZE="50" NAME="file" VALUE="/public_html/">]
    print [<BR><BR>]
    print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
    print [</FORM>]
    quit
]
```

```
; check user/pass, end program if incorrect:

username: submitted/2 password: submitted/4
either (username = "user") and (password = "pass") [
    ; if user/pass is ok, go on
][
    print "content-type: text/html^/"
    print "Incorrect Username/Password." quit
]

print rejoin [
    "Content-Type: application/x-unknown"
    newline
    "Content-Length: "
    (size? to-file join root-path submitted/6)
    newline
    "Content-Disposition: attachment; filename="
    (second split-path to-file submitted/6)
    newline
]

data: read/binary to-file join root-path submitted/6
data-length: size? to-file join root-path submitted/6
write-io system/ports/output data data-length
```

9.9.5 A Complete Web Server Management Application

This final script makes use of several previous examples, and some additional code, to form a complete web server management application. It allows you to list directory contents, upload, download, edit, and search for files, execute OS commands (chmod, ls, mv, cp, etc. - any command available on your web server's operating system), and run REBOL commands directly on your server. Edited files are automatically backed up into an "edit_history" folder on the server before being saved. No configuration is required for most web servers. Just save this script as "web-tool.cgi", upload it *and* the REBOL interpreter into the same folder as your web site's index.html file, set permissions (chmod) to 755, then go to http://yourwebsite/web-tool.cgi.

THIS SCRIPT CAN POSE A MAJOR SECURITY THREAT TO YOUR SERVER. It can potentially enable anyone to gain control of your web server and everything it contains. DO NOT install it on your server if you're at all concerned about security, or if you don't know how to secure your server yourself.

The first line of this script must point to the location of the REBOL interpreter on your web server, and you must use a version of REBOL which supports the "call" function (version 2.76 is recommended). By default, the REBOL interpreter should be uploaded to the same path as this script, that folder should be publicly accessible, and you must *upload the correct version of REBOL for the operating system on which your server runs.* IN THIS EXAMPLE, THE REBOL INTERPRETER HAS BEEN RENAMED "REBOL276".

```
#! ./rebol276 -cs
REBOL [Title: "REBOL CGI Web Site Manager"]

;-----------------------------------------------------------------------
; Upload this script to the same path as index.html on your server, then
; upload the REBOL interpreter to the path above (the same path as the
; script, by default).  CHMOD IT AND THIS SCRIPT TO 755.  Then, to run the
; program, go to www.yoursite.com/this-script.cgi .
;-----------------------------------------------------------------------

; YOU CAN EDIT THESE VARIABLES, _IF_ NECESSARY (change the quoted values):
```

```
; The user name you want to use to log in:

    set-username:   "username"

; The password you want to use to log in:

    set-password:   "password"

;-------------------------------------------------------------------------

; Do NOT edit these variables, unless you really know what you're doing:

doc-path: to-string what-dir
script-subfolder: find/match what-dir doc-path
if script-subfolder = none [script-subfolder: ""]

;-------------------------------------------------------------------------

; Get submitted data:

selection: decode-cgi system/options/cgi/query-string

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    the-data: data
    data
]
submitted: read-cgi
submitted-block: decode-cgi the-data

; -------------------------------------------------------------------------

; This section should be first because it prints a different header
; for a push download (not "content-type: text/html^/"):

if selection/2 = "download-confirm" [
    print rejoin [
        "Content-Type: application/x-unknown"
        newline
        "Content-Length: "
        (size? to-file selection/4)
        newline
        "Content-Disposition: attachment; filename="
        (second split-path to-file selection/4)
        newline
    ]

    data: read/binary to-file selection/4
    data-length: size? to-file selection/4
    write-io system/ports/output data data-length
    quit
]
```

```
;-------------------------------------------------------------------------

; Print the normal HTML headers, for use by the rest of the script:

print "content-type: text/html^/"
print {<HTML><HEAD><TITLE>Web Site Manager</TITLE></HEAD><BODY>}

;-------------------------------------------------------------------------

; If search has been called (via link on main form):

if selection/2 = "confirm-search" [
    print rejoin [
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {<center><table border="1" cellpadding="10" width=80%><tr><td>}
    print [<CENTER><TABLE><TR><TD>]
    print rejoin [
        {<FORM ACTION="./} (second split-path system/options/script)
        {"> Text to search for: <BR> <INPUT TYPE="TEXT" SIZE="50"}
        {NAME="phrase"><BR><BR>Folder to search in: <BR>}
        {<INPUT TYPE="TEXT" SIZE="50" NAME="folder" VALUE="} what-dir
        {" ><BR><BR><INPUT TYPE=hidden NAME=perform-search }
        {VALUE="perform-search"><INPUT TYPE="SUBMIT" NAME="Submit" }
        {VALUE="Submit"></FORM></TD></TR></TABLE></CENTER>}
        {</td></tr></table></center></BODY></HTML>}
    ]
    quit
]

;-------------------------------------------------------------------------

; If edited file text has been submitted:

if submitted-block/2 = "save" [

    ; Save newly edited document:
    write (to-file submitted-block/6) submitted-block/4
    print {<center><strong>Document Saved:</strong>
        <br><br><table border="1" width=80% cellpadding="10"><tr><td>}
    prin [<center><textarea cols="100" rows="15" name="contents">]
    prin replace/all read (
        to-file (replace/all submitted-block/6 "%2F" "/")
    ) "</textarea>" "<\/textarea>"
    print [</textarea></center>]
    print rejoin [
        {</td></tr></table><br><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
        {</BODY></HTML>}
    ]
    quit
]

;-------------------------------------------------------------------------

; If upload link has been clicked, print file upload form:

if selection/2 = "upload-confirm" [
    print rejoin [
```

```
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {<center><table border="1" cellpadding="10" width=80%><tr><td>}
    print {<center>}

    ; If just the link was clicked - no data submitted yet:

    if selection/4 = none [
        print rejoin [
            {<FORM ACTION="./} (second split-path system/options/script)
            {" METHOD="post" ENCTYPE="multipart/form-data">
                <strong>Upload File:</strong><br><br>
                <INPUT TYPE=hidden NAME=upload-confirm
                VALUE="upload-confirm">
                <INPUT TYPE="file" NAME="photo"> <br><br>
                Folder: <INPUT TYPE="text" NAME="path" SIZE="35"
                    VALUE="} what-dir {">
                <INPUT TYPE="submit" NAME="Submit" VALUE="Upload">
            </FORM>
            <br></center></td></tr></table></center></BODY></HTML>}
        ]
        quit
    ]
]

;-------------------------------------------------------------------------

; If upload data has been submitted:

if (submitted/2 = #"-") and (submitted/4 = #"-") [

    ; This function is by Andreas Bolka:

    decode-multipart-form-data: func [
        p-content-type
        p-post-data
        /local list ct bd delim-beg delim-end non-cr
        non-lf non-crlf mime-part
    ] [
        list: copy []
        if not found? find p-content-type "multipart/form-data" [
            return list
        ]
        ct: copy p-content-type
        bd: join "--" copy find/tail ct "boundary="
        delim-beg: join bd crlf
        delim-end: join crlf bd
        non-cr:     complement charset reduce [ cr ]
        non-lf:     complement charset reduce [ newline ]
        non-crlf:   [ non-cr | cr non-lf ]
        mime-part:  [
            ( ct-dispo: content: none ct-type: "text/plain" )
            delim-beg ; mime-part start delimiter
            "content-disposition: " copy ct-dispo any non-crlf crlf
            opt [ "content-type: " copy ct-type any non-crlf crlf ]
            crlf ; content delimiter
            copy content
            to delim-end crlf ; mime-part end delimiter
            ( handle-mime-part ct-dispo ct-type content )
        ]
        handle-mime-part: func [
```

```
            p-ct-dispo
            p-ct-type
            p-content
            /local tmp name value val-p
        ] [
            p-ct-dispo: parse p-ct-dispo {;="}
            name: to-set-word (select p-ct-dispo "name")
            either (none? tmp: select p-ct-dispo "filename")
                   and (found? find p-ct-type "text/plain") [
                value: content
            ] [
                value: make object! [
                    filename: copy tmp
                    type: copy p-ct-type
                    content: either none? p-content [none][copy p-content]
                ]
            ]
            either val-p: find list name
                [
                    change/only next val-p compose [
                        (first next val-p) (value)
                    ]
                ]
                [append list compose [(to-set-word name) (value)]]
        ]
        use [ct-dispo ct-type content] [
            parse/all p-post-data [some mime-part "--" crlf]
        ]
        list
    ]

    ; After the following line, "probe cgi-object" will display all parts
    ; of the submitted multipart object:

    cgi-object: construct decode-multipart-form-data
        system/options/cgi/content-type copy submitted

    ; Write file to server using the original filename, and notify the
    ; user:

    the-file: last split-path to-file copy cgi-object/photo/filename
    write/binary
        to-file join cgi-object/path the-file
        cgi-object/photo/content
    print rejoin [
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {
        <center><table border="1" width=80% cellpadding="10"><tr><td>
        <strong>UPLOAD COMPLETE</strong><br><br></center>
        <strong>Files currently in this folder:</strong><br><br>
    }
    change-dir to-file cgi-object/path
    folder: sort read what-dir
    foreach file folder [
        print [
            rejoin [
                {<a href="./} (second split-path system/options/script)
                {?editor-confirm=editor-confirm&thefile=}
                what-dir file {">(edit)</a>   }
                {<a href="./} (second split-path system/options/script)
```

```
                {?download-confirm=download-confirm&thefile=}
                what-dir file {">} "(download)</a>   "
                {<a href="./} (find/match what-dir doc-path) file
                {">} file {</a><br>}
            ]
        ]
    ]
    print {</td></tr></table></center></BODY></HTML>}
    quit
]

;-------------------------------------------------------------------------

; If no data has been submitted, print form to request user/pass:

if ((selection/2 = none) or (selection/4 = none)) [
    print rejoin [{
        <STRONG>W A R N I N G  -  Private Server, Login Required:</STRONG>
        <BR><BR>
        <FORM ACTION="./} (second split-path system/options/script) {">
        Username: <INPUT TYPE=text SIZE="50" NAME="name"><BR><BR>
        Password: <INPUT TYPE=text SIZE="50" NAME="pass"><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM></BODY></HTML>
    }]
    quit
]

;-------------------------------------------------------------------------

; If a folder name has been submitted, print file list:

if ((selection/2 = "command-submitted") and (
    selection/4 = "call {^/^/^/^/}")
) [
    print rejoin [
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {<center><table border="1" cellpadding="10" width=80%><tr><td>}
    print {<strong>Files currently in this folder:</strong><br><br>}
    change-dir to-file selection/6
    folder: sort read what-dir
    foreach file folder [
        print rejoin [
            {<a href="./} (second split-path system/options/script)
            {?editor-confirm=editor-confirm&thefile=}
            what-dir file {">} "(edit)</a>   "
            {<a href="./} (second split-path system/options/script)
            {?download-confirm=download-confirm&thefile=}
            what-dir file {">} "(download)</a>   "
            {<a href="./} (find/match what-dir doc-path) file {">} file
            {</a><br>}
        ]
    ]
    print {</td></tr></table></center></BODY></HTML>}
    quit
]

;-------------------------------------------------------------------------

; If editor has been called (via a constructed link):
```

```
if selection/2 = "editor-confirm" [

    ; backup (before changes are made):

    cur-time: to-string replace/all to-string now/time ":" "-"
    document_text: read to-file selection/4
    if not exists? to-file rejoin [
        doc-path script-subfolder "edit_history/"
    ] [
        make-dir to-file rejoin [
            doc-path script-subfolder "edit_history/"
        ]
    ]
    write to-file rejoin [
        doc-path script-subfolder "edit_history/"
        to-string (second split-path to-file selection/4)
        "--" now/date "_" cur-time ".txt"
    ] document_text

    ; note the POST method in the HTML form:

    print rejoin [
        {<center><strong>Be sure to SUBMIT when done:</strong>}
        {<BR><BR><FORM method="post" ACTION="./}
        (second split-path system/options/script) {">}
        {<INPUT TYPE=hidden NAME=submit_confirm VALUE="save">}
        {<textarea cols="100" rows="15" name="contents">}
        (replace/all document_text "</textarea>" "<\/textarea>")
        {</textarea><BR><BR><INPUT TYPE=hidden NAME=path VALUE="}
        selection/4
        {"><INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM></center></BODY></HTML>}
    ]
    quit
]

;-------------------------------------------------------------------------

; If search criteria has been entered:

if selection/6 = "perform-search" [
    phrase: selection/2
    start-folder: to-file selection/4
    change-dir start-folder
    ; found-list: ""

    recurse: func [current-folder] [
        foreach item (read current-folder) [
            if not dir? item [
                if error? try [
                    if find (read to-file item) phrase [
                        print rejoin [
                            {<a href="./}
                            (second split-path system/options/script)
                            {?editor-confirm=editor-confirm&theitem=}
                            what-dir item {">(edit)</a>   }
                            {<a href="./}
                            (second split-path system/options/script)
                            {?download-confirm=download-confirm&theitem=}
                            what-dir item {">(download)</a>   "}
                            phrase {" found in:  }
                            {<a href="./} (find/match what-dir doc-path)
```

```
                            item {">} item {</a><BR>}
                        ]
                        ; found-list: rejoin [
                        ;     found-list newline what-dir item
                        ; ]
                    ]
                ] [print rejoin ["error reading " item]]
            ]
        ]
        foreach item (read current-folder) [
            if dir? item [
                change-dir item
                recurse %.\
                change-dir %..\
            ]
        ]
    ]

    print rejoin [
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {<center><table border="1" cellpadding="10" width=80%><tr><td>}
    print rejoin [
        {<strong>SEARCHING for "} phrase {" in } start-folder
        {</strong><BR><BR>}
    ]
    recurse %.\
    print {<BR><strong>DONE</strong><BR>}
    print {</td></tr></table></center></BODY></HTML>}
    ; save %found.txt found-list
    quit
]

;-------------------------------------------------------------------------

; This is the main entry form, used below:

entry-form: [
    print rejoin [
        {<CENTER><strong>current path: </strong>} what-dir
        {<FORM METHOD="get" ACTION="./}
        (second split-path system/options/script) {">}{<INPUT TYPE=hidden}
        { NAME=submit_confirm VALUE="command-submitted">}
        {<TEXTAREA COLS="100" ROWS="10" NAME="contents">}
        {call {^/^/^/^/}</textarea><BR><BR>}
        {List Files: <INPUT TYPE=text SIZE="35" NAME="name" VALUE="}
        what-dir {"><INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">}
        {          <A HREF="./} (second split-path system/options/script)
        {?upload-confirm=upload-confirm">upload</A>       } ; leave spaces
        {<A HREF="./} (second split-path system/options/script)
        {?confirm-search=confirm-search">search</A>}
        {</FORM><BR></CENTER>}
    ]
]

;-------------------------------------------------------------------------

; If code has been submitted, print the output, along with an entry form):

if ((selection/2 = "command-submitted") and (
selection/4 <> "call {^/^/^/^/}") and ((to-file selection/6) = what-dir))[
```

```
    write %commands.txt join "REBOL[]^/" selection/4
    ; The "call" function requires REBOL version 2.76:
    call/output/error
        "./rebol276 -qs commands.txt"
        %conso.txt %conse.txt
    do entry-form
    print rejoin [
        {<CENTER>Output: <BR><BR>}
        {<TABLE WIDTH=80% BORDER="1" CELLPADDING="10"><TR><TD><PRE>}
        read %conso.txt
        {</PRE></TD></TR></TABLE><BR><BR>}
        {Errors: <BR><BR>}
        read %conse.txt
        {</CENTER></BODY></HTML>}
    ]
    quit
]
;-------------------------------------------------------------------------

if ((selection/2 = "command-submitted") and (
    selection/4 <> "call {^/^/^/^/}") and (
    (to-file selection/6) <> what-dir)
) [
    print rejoin [
        {<center><a href="./}
        (second split-path system/options/script) {?name=} set-username
        {&pass=} set-password {">Back to Web Site Manager</a></center>}
    ]
    print {
        <center><table border="1" cellpadding="10" width=80%><tr><td>
            <center>
        You must EITHER enter a command, OR enter a file path to list.<BR>
        Please go back and try again (refresh the page if needed).
            </center>
        </td></tr></center></BODY></HTML>
    }
    quit
]

;-------------------------------------------------------------------------

; Otherwise, check submitted user/pass, then print form for code entry:

username: selection/2 password: selection/4
either (username = set-username) and (password = set-password) [
    ; if user/pass is ok, go on
][
    print "Incorrect Username/Password. </BODY></HTML>" quit
]

do entry-form
print {</BODY></HTML>}
```

Be sure to see the following links for more insight about REBOL CGI programming:

http://rebol.com/docs/cgi1.html
http://rebol.com/docs/cgi2.html
http://rebol.com/docs/cgi-bbs.html
http://www.rebol.net/cookbook/recipes/0045.html

To create web sites using a PHP-like version of REBOL that runs in a web server written entirely in REBOL, see:

http://cheyenne-server.org (binaries are available for Windows, Mac, and Linux).
http://cheyenne-server.org/docs/rsp-api.html - documentation for the "RSP" (REBOL server pages) API.

9.10 WAP - Cell Phone Browser CGI Apps

Most cell phone service providers offer data options which allow users to access information on the Internet. If you use a "smart phone", your data package likely allows you to access normal web pages using a browser program that runs on your phone (with varying degrees of rendering success). Most average cell phones, however, come only with a "WAP" browser that allows you to access only very light mobile versions of web sites, and provides information in a format specifically accessible only by cell phones. Using WAP mobile sites, you can check email in Google, Yahoo, and other accounts, read news, get weather and traffic reports, manage Ebay transactions, etc. WAP versions of sites, accessible on normal cell phones are not renditions of the normal HTML sites created or interpreted by your phone, but are instead entirely separate versions of each site, created and managed on the web server by the web site creators, and simply accessed by WAP phone browsers.

You can create your own WAP CGI applications, to be accessed by any phone with a WAP browser, using REBOL. WAP scripts are just as easy to ceate as normal CGI web scripts. Instead of HTML, however, you simply need to format the output of your scripts using WAP ("Wireless Application Protocal") syntax. Reference information about WAP tags ("WML") can be found at http://www.w3schools.com/WAP/wml_reference.asp

Here's a basic WAP CGI script which demonstrates the stock code required in all your REBOL WAP scripts. The first 5 lines should be memorized. You'll need to copy them verbatim to the beginning of every WAP script you create. The last lines demonstrate some other important WAP tags. Notice that the wml tag basically replaces the html tag from normal HTML. Every WAP page also contains card tags, which basically replace the head and body tags in normal HTML. This script prints the current time in a WAP browser. *Remember to set the permissions of (chmod) any WAP CGI script to 755*:

```
#!./rebol276 -cs
REBOL []
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

; The lines above are standard headers for any WAP CGI script.  The
; lines below print out the standard tags needed to print text on a
; WAP page.  Included inside the wml, card and p (paragraph) tags is
; one line that prints the current time:

print {<wml><card id="1" title="Current Time"><p>}
print now
print {</p></card></wml>}
quit
```

The following nearly identical script converts text data output by another script on the web site to WAP format, so that it can be read on small cell phone browsers. Because WAP syntax is a bit more strict than HTML, some HTML tags must be removed (replaced) from the source script output. You must be careful to strip out unnecessary tags and characters in text formatted for display in cell phones. Most WAP browsers will simply display an error if they encounter improperly formatted content:

```
#!./rebol276 -cs
REBOL []
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}
```

```
print {<wml><card id="1" title="Wap Page"><p>}
prin replace/all (read http://site.com/page.cgi) {</BODY> </HTML>} {}
print {</p></card></wml>}
quit
```

Here's a bit more useful version of the above script which allows users to specify the file to be converted, right in the URL of WAP page (i.e., if this script is at site.com/wap.cgi, and the user wants to read page.txt in their WAP browser, then the URL would be "http://site.com/wap.cgi?page.txt"):

```
#!./rebol276 -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}
print {<wml><card id="1" title="Wap Page"><p>}

; The line below joins the web site URL with the submitted page,
; reads it, and parses it, up to some indicated marker text, so
; that only the text before the marker text is displayed:

parse read join http://site.com/ submitted/2 [
    thru submitted/2 copy p to "some marker text"
]
prin p

print {</p></card></wml>}
quit
```

Here's a version of the above script that lets users select a document to be converted and displayed. This code makes use of select and option tags, which work like HTML dropdown boxes in forms. It also demonstrates how to use anchor, go and postfield tags to submit form data. These work like the "action" in HTML forms. Variable labels for information entered by the user are preceded by a dollar symbol ("$"), and the variable name is enclosed in parentheses (i.e., the $(doc) variable below is submitted to the wap.cgi program). The anchor tag creates a clickable link that makes the action occur:

```
#!./rebol -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

; If no data has been submitted, do this by default:

if submitted/2 = none [

    ; Print some standard tags:

    print {<wml><card id="1" title="Select Doc"><p>}

    ; Get a list of subfolders and display them in a dropdown box:

    folders: copy []
    foreach folder read %./docs/ [
        if find to-string folder {/} [append folders to-string folder]
```

```
    ]
    print {Doc: <select name="doc">}
    foreach folder folders [
        folder: replace/all folder {/} {}
        print rejoin [{<option value="} folder {">} folder {</option>}]
    ]
    print {</select>

    ; Create a link to submit the chosen folder, then close the tags
    ; from above:

    <anchor>
       <go method="get" href="wap.cgi">
           <postfield name="doc" value="$(doc)"/>
       </go>
       Submit
    </anchor>}
    print {</p></card></wml>}
    quit
]

; If some data has been submitted, read the selected doc:

print rejoin [{<wml><card id="1" title="} submitted/2 {"><p>}]

parse read join http://site.com/docs/ submitted/2 [
    thru submitted/2 copy p to "end of file"
]
prin p

print {</p></card></wml>}
quit
```

This script breaks up the selected text document into small (130 byte) chunks so that it can be navigated more quickly. Each separate card is displayed as a different page in the WAP browswer, and anchor links are provided to navigate forward and backword between the cards. For the user, paging through data in this way tends to be *much* faster than scrolling through long results line by line:

```
#!./rebol276 -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

count: 0
p: read http://site.com/folder.txt
print {<wml>}
forskip p 130 [

    ; Create a counter, to appear as each card title, then print links
    ; to go forward and backward between the cards:

    count: count + 1
    print rejoin [{<card id="} count {" title="page-} count {"><p>}]
    print rejoin [
        {<anchor>Next<go href="#} (count + 1) {"/></anchor>}
    ]
    print rejoin [{<anchor>Back<prev/></anchor>}]
```

```
    ; Print 130 characters in each card:

    print copy/part p 130
    print {</p></card>}
]
print {</wml>}
quit
```

This next script combines the techniques explained so far, and allows the user to *select a file* on the web server, using a dropdown box, and displays the selected file in 130 byte pages:

```
#!./rebol276 -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

if submitted/2 = none [
    print {<wml><card id="1" title="Select Teacher"><p>}
    ; print {Name: <input name="name" size="12"/>}
    folders: copy []
    foreach folder read %./Teachers/ [
        if find to-string folder {/} [append folders to-string folder]
    ]
    print {Teacher: <select name="teacher">}
    foreach folder folders [
        folder: replace/all folder {/} {}
        print rejoin [
            {<option value="} folder {">} folder {</option>}
        ]
    ]
    print {</select>
    <anchor>
       <go method="get" href="wap.cgi">
           <postfield name="teacher" value="$(teacher)"/>
       </go>
       Submit
    </anchor>}
    print {</p></card></wml>}
    quit
]

count: 0
parse read join http://site.com/folder/ submitted/2 [
    thru submitted/2 copy p to "past students"
]
print {<wml>}

forskip p 130 [
    count: count + 1
    print rejoin [
        {<card id="} count {" title="} submitted/2 "-" count {"><p>}
    ]
    print rejoin [
        {<anchor>Next<go href="#} (count + 1) {"/></anchor>}
    ]
    print rejoin [{<anchor>Back<prev/></anchor>}]
    print copy/part p 130
```

```
    print {</p></card>}
]
print {</wml>}
quit
```

Finally, this script allows users to select a file, and enter some text to be saved in that file, using the input tag:

```
#!./rebol276 -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

if submitted/2 = none [
    print {<wml><card id="1" title="Select Teacher"><p>}
    print {Insert Text: <input name="thetext" size="12"/>}
    folders: copy []
    foreach folder read %./Teachers/ [
        if find to-string folder {/} [
            append folders to-string folder
        ]
    ]
    print {Teacher: <select name="teacher">}
    foreach folder folders [
        folder: replace/all folder {/} {}
        print rejoin [
            {<option value="} folder {">} folder {</option>}
        ]
    ]
    print {</select>
    <anchor>
       <go method="get" href="wapinsert.cgi">
           <postfield name="teacher" value="$(teacher)"/>
           <postfield name="thetext" value="$(thetext)"/>
       </go>
       Submit
    </anchor>}
    print {</p></card></wml>}
    quit
]

chosen-file: rejoin [%./Teachers/ submitted/2 "/schedule.txt"]
adjusted-file: read/lines chosen-file
insert next next next next adjusted-file submitted/4
write/lines chosen-file adjusted-file

count: 0
parse read join http://site.com/folders/ submitted/2 [
    thru submitted/2 copy p to "past students"
]
print {<wml>}

forskip p 130 [
    count: count + 1
    print rejoin [
        {<card id="} count {" title="} submitted/2 "-" count {"><p>}
    ]
    print rejoin [
```

```
        {<anchor>Next<go href="#} (count + 1) {"/></anchor>}
    ]
    print rejoin [{<anchor>Back<prev/></anchor>}]
    print copy/part p 130
    print {</p></card>}
]
print {</wml>}
quit
```

This script allows users to read email messages from any POP server, on their cell phone:

```
#!./rebol276 -cs
REBOL []
submitted: decode-cgi system/options/cgi/query-string
prin {Content-type: text/vnd.wap.wml^/^/}
prin {<?xml version="1.0" encoding="iso-8859-1"?>^/}
prin {<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">^/}

accounts: [
    ["pop.server" "smtp.server" "username" "password" you@site.com]
    ["pop.server2" "smtp.server2" "username" "password" you@site2.com]
    ["pop.server3" "smtp.server3" "username" "password" you@site3.com]
]

if ((submitted/2 = none) or (submitted/2 = none)) [
    print {<wml><card id="1" title="Select Account"><p>}
    print {Account: <select name="account">}
    forall accounts [
        print rejoin [
            {<option value="} index? accounts {">}
            last first accounts {</option>}
        ]
    ]
    print {</select>
    <select name="readorsend">
        <option value="readselect">Read</option>
        <option value="sendinput">Send</option>
    </select>
    <anchor>
       <go method="get" href="wapmail.cgi">
           <postfield name="account" value="$(account)"/>
           <postfield name="readorsend" value="$(readorsend)"/>
       </go>
       Submit
    </anchor>}
    print {</p></card></wml>}
    quit
]

if submitted/4 = "readselect" [
    t: pick accounts (to-integer submitted/2)
    system/schemes/pop/host:  t/1
    system/schemes/default/host: t/2
    system/schemes/default/user: t/3
    system/schemes/default/pass: t/4
    system/user/email: t/5
    prin {<wml><card id="1" title="Choose Message"><p>}
    prin rejoin [{<setvar name="account" value="} submitted/2 {"/>}]
    prin {<select name="chosenmessage">}
    mail: read to-url join "pop://" system/user/email
```

```
    foreach message mail  [
        pretty: import-email message
        if (find pretty/subject "***SPAM***") = none [
            replace/all pretty/subject {"} {}
            replace/all pretty/subject {&} {}
            prin rejoin [
                {<option value="}
                pretty/subject
                {">}
                pretty/subject
                {</option>}
            ]
        ]
    ]
    print {</select>
    <anchor>
       <go method="get" href="wapmail.cgi">
           <postfield name="subroutine" value="display"/>
           <postfield name="chosenmessage" value="$(chosenmessage)"/>
           <postfield name="account" value="$(account)"/>
       </go>
       Submit
    </anchor>
    </p></card></wml>}
    quit
]

if submitted/2 = "display" [
    t: pick accounts (to-integer submitted/6)
    system/schemes/pop/host:  t/1
    system/schemes/default/host: t/2
    system/schemes/default/user: t/3
    system/schemes/default/pass: t/4
    system/user/email: t/5
    prin {<wml><card id="1" title="Display Message"><p>}
    mail: read to-url join "pop://" system/user/email
    foreach message mail  [
        pretty: import-email message
        if pretty/subject = submitted/4 [
            replace/all pretty/content {"} {}
            replace/all pretty/content {&} {}
            replace/all pretty/content {3d} {}
            prin pretty/content
        ]
    ]
    print {</p></card></wml>}
    quit
]
```

Creating WAP versions of your REBOL CGI scripts is a fantastic way to provide even more universal access to your important data.

9.11 REBOL as a Browser Plugin

REBOL interpreters exist not only for an enormous variety of operating systems, but also as plugins for several popular browsers (Internet Explorer and many Mozilla variations, including Opera). That means that you can embed the REBOL interpreter directly into a web page, and have complete, complex REBOL programs run right *inside* pages of your web site (in a way similar to Flash and Java, and useful like Javascript code). This provides a nice alternative to CGI programming for any type of application that runs appropriately in the stand-alone view.exe interpreter (i.e., for games, multimedia applications, and rich graphic/GUI applications of all types). Since the browser plugin runs typical REBOL code in the same way as the downloadable view.exe interpreter, you can run code directly on your web pages, without making any changes.

To use the plugin on a web page, just include the necessary object code on your page, as in the following example. Be sure to change the URL of the script you want to run, the size of the window you want to create, and other parameters as needed. You can download the rebolb7.cab file, upload it to your own site, and run it from there, if you prefer (be aware that there are several different versions: "http://www.rebol.com/plugin/rebolb5.cab#Version=0,5,0,0", "http://www.rebol.com/plugin/rebolb6.cab#Version=0,6,0,0", and "http://www.rebol.com/plugin/rebolb7.cab#Version=1,0,0,0" - use version 7 unless you have a specific reason not to). If you use your own copy, be sure to change the URL of the cab file in the following example:

```
<HTML><HEAD><TITLE>REBOL Plugin Test</TITLE></HEAD><BODY><CENTER>

<OBJECT ID="REBOL" CLASSID="CLSID:9DDFB297-9ED8-421d-B2AC-372A0F36E6C5"
    CODEBASE="http://www.rebol.com/plugin/rebolb7.cab#Version=1,0,0,0"
    WIDTH="500" HEIGHT="400">
    <PARAM NAME="LaunchURL" VALUE="http://yoursite.com/test_plugin.r">
    <PARAM NAME="BgColor" VALUE="#FFFFFF">
    <PARAM NAME="Version" VALUE="#FFFFFF">
    <PARAM NAME="BgColor" VALUE="#FFFFFF">
</OBJECT>

</CENTER></BODY></HTML>
```

Here's an example script that could be run at the "http://yoursite.com/test_plugin.r" link referenced in the code above. You can replace this with *any* valid REBOL code, and have it run directly in your browser:

```
REBOL []
view layout [
    size 500x400
    btn "Click me" [alert "Plugin is working!"]
]
```

You can see the above example running at http://re-bol.com/test_plugin.html. The above script must be run in Internet Explorer on MS Windows. In order to use the plugin in Mozilla based browsers (Firefox, Opera, etc.), the following code must be pasted into your HTML page:

```
<HTML><HEAD><TITLE>REBOL Plugin Test</TITLE></HEAD><BODY><CENTER>

<OBJECT ID="REBOL_IE" CLASSID="CLSID:9DDFB297-9ED8-421d-B2AC-372A0F36E6C5"
    CODEBASE="http://re-bol.com/rebolb7.cab#Version=1,0,0,0"
    WIDTH="500" HEIGHT="400" BORDER="1" ALT="REBOL/Plugin">
    <PARAM NAME="bgcolor" value="#ffffff">
    <PARAM NAME="version" value="1.2.0">
    <PARAM NAME="LaunchURL" value="http://re-bol.com/test_plugin.r">
    <embed name="REBOL_Moz" type="application/x-rebol-plugin-v1"
```

```
        WIDTH="500" HEIGHT="400" BORDER="1" ALT="REBOL/Plugin"
        bgcolor="#ffffff"
        version="1.2.0"
        LaunchURL="http://re-bol.com/test_plugin.r"
    >
    </embed>
</OBJECT>

<script language="javascript" type="text/javascript">
var plugin_name = "REBOL/Plugin for Mozilla";
function install_rebol_plugin_mozilla() {
    if (navigator.userAgent.toLowerCase().indexOf("msie") == -1) {
        if (InstallTrigger) {
           xpi={'REBOL/Plugin for Mozilla':'http://re-bol.com/mozb2.xpi'};
           InstallTrigger.install(xpi, installation_complete);
        }
    }
}
function installation_complete(url, status) {
    if (status == 0) location.reload();
}
function is_mozilla_plugin_installed() {
    if (window.navigator.plugins) {
        for (var i = 0; i < window.navigator.plugins.length; i++) {
            if (window.navigator.plugins[i].name == plugin_name)
            return true;
        }
    }
    return false;
}
if (!is_mozilla_plugin_installed()) install_rebol_plugin_mozilla();
</script>

</CENTER></BODY></HTML>
```

You can see the above script working at http://re-bol.com/test_plugin_mozilla.html.

For more information about the REBOL plugins, see http://www.rebol.com/plugin/install.html and http://home.comcast.net/~rebolore/plugin-guide.pdf.

9.12 Using Databases

Databases manage all the difficult details of searching, sorting, and otherwise manipulating large amounts of data, quickly and safely in a multiuser environment. MySQL is a free, open source database system used in many web sites and software projects. ODBC is a common interface that allows programmers to connect to many other types of databases. The most recent releases of REBOL, along with all commercial versions, have built-in native access to MySQL, ODBC, and other database formats. You can also download a free MySQL protocol module that runs in *every* free version of REBOL, from http://softinnov.org/rebol/mysql.shtml. A free module for the postgre database system is also available at that site.

To explore database concepts and techniques in this section, we'll use the open source MySQL module above because it provides access to a powerful and popular database solution which works even in old and unusual versions of REBOL.

Most web hosting accounts come with MySQL already installed. See your web host account instructions to learn how to access it (you need a web address, database name, user name, and password). To get a free, simple-to-install web server package for Windows that includes MySQL, go to http://www.uniformserver.com. That program enables you to easily install a web server with MySQL pre-configured on your local computer. It's useful if you don't have access to a web server on the Internet, or if you want to create multiuser applications which use MySQL on a local network.

To use the REBOL MySQL module above, unpack the compressed "rip" file available at the link above. This step only needs to be done the first time you use the package on a given computer:

```
do mysql-107.rip
```

Every time you access a MySQL database, you need to "do" the module that was unpacked in the step above:

```
do %mysql-r107/mysql-protocol.r

; At the time of this writing, 1.07 was the most current version of
; the mysql module.  Update the numbers in the previous two lines
; of code to reflect the current version number you've downloaded.
```

Next, enter the database info (location, username, and password), as in the instructions at http://softinnov.org/rebol/mysql-usage.html:

```
db: open mysql://username:password@yourwebsite.com/yourdatabasename
```

In MySQL and other databases, data is stored in "tables". Tables are made up of columns of related information. A "Contacts" table, for example, may contain name, address, phone, and birthday columns. Each entry in the database can be thought of as a row containing info in each of those column fields:

```
name            address                 phone       birthday
----            -------                 --------    --------
John Smith      123 Toleen Lane         555-1234    1972-02-01
Paul Thompson   234 Georgetown Place    555-2345    1972-02-01
Jim Persee      345 Portman Pike        555-3456    1929-07-02
George Jones    456 Topforge Court                  1989-12-23
Tim Paulson                             555-5678    2001-05-16
```

"SQL" statements let you work with data stored in the table. Some SQL statements are used to

create, destroy, and fill columns with data:

```
CREATE TABLE table_name          ; create a new table of information
DROP TABLE table_name            ; delete a table
INSERT INTO table_name VALUES (value1, value2,....)
INSERT INTO Contacts
    VALUES ('Billy Powell', '5 Binlow Dr.', '555-6789', '1968-04-19')
INSERT INTO Contacts (name, phone)
    VALUES ('Robert Ingram', '555-7890')
```

The SELECT statement is used to retrieve information from columns in a given table:

```
SELECT column_name(s) FROM table_name
SELECT * FROM Contacts
SELECT name,address FROM Contacts
SELECT DISTINCT birthday FROM Contacts ; returns no duplicate entries
; To perform searches, use WHERE.  Enclose search text in single
; quotes and use the following operators:
;  =, <>, >, <, >=, <=, BETWEEN, LIKE (use "%" for wildcards)
SELECT * FROM Contacts WHERE name='John Smith'
SELECT * FROM Contacts WHERE name LIKE 'J%'
SELECT * FROM Contacts WHERE birthday LIKE '%72%' OR phone LIKE '%34'
SELECT * FROM Contacts
    WHERE birthday NOT BETWEEN '1900-01-01' AND '2010-01-01'
; IN lets you specify a list of data to match within a column:
SELECT * FROM Contacts WHERE phone IN ('555-1234','555-2345')
SELECT * FROM Contacts ORDER BY name  ; sort results alphabetically
SELECT name, birthday FROM Contacts ORDER BY birthday, name DESC
```

Other SQL statements:

```
UPDATE Contacts SET address = '643 Pine Valley Rd.'
    WHERE name = 'Robert Ingram'     ; alter or add to existing data
DELETE FROM Contacts WHERE name = 'John Smith'
DELETE * FROM Contacts
ALTER TABLE  - change the column structure of a table
CREATE INDEX - create a search key
DROP INDEX   - delete a search key
```

To integrate SQL statements in your REBOL code, enclose them as follows:

```
insert db {SQL command}
```

To retrieve the result set created by an inserted command, use:

```
copy db
```

You can use the data results of any query just as you would any other data contained in REBOL blocks. To retrieve only the first result of any command, for example, use:

```
first db
```

When you're done using the database, close the connection:

```
close db
```

Here's a complete example that opens a database connection, creates a new "Contacts" table, inserts data into the table, makes some changes to the table, and then retrieves and prints all the contents of the table, and closes the connection:

```
REBOL []

do %mysql-protocol.r
db: open mysql://root:root@localhost/Contacts
; insert db {drop table Contacts} ; erase the old table if it exists
insert db {create table Contacts (
    name            varchar(100),
    address         text,
    phone           varchar(12),
    birthday        date
)}
insert db {INSERT into Contacts VALUES
    ('John Doe', '1 Street Lane', '555-9876', '1967-10-10'),
    ('John Smith', '123 Toleen Lane', '555-1234', '1972-02-01'),
    ('Paul Thompson', '234 Georgetown Pl.', '555-2345', '1972-02-01'),
    ('Jim Persee', '345 Portman Pike', '555-3456', '1929-07-02'),
    ('George Jones', '456 Topforge Court', '', '1989-12-23'),
    ('Tim Paulson', '', '555-5678', '2001-05-16')
}
insert db "DELETE from Contacts WHERE birthday = '1967-10-10'"
insert db "SELECT * from Contacts"
results: copy db
probe results
close db
halt
```

Here's a shorter coding format that can be used to work with database tables quickly and easily:

```
read join mysql://user:pass@host/DB? "SELECT * from DB"
```

For example:

```
foreach row read rejoin [mysql://root:root@localhost/Contacts?
    "SELECT * from Contacts"] [print row]
```

Here's a GUI example:

```
results: read rejoin [
    mysql://root:root@localhost/Contacts? "SELECT * from Contacts"]
view layout [
    text-list 100x400 data results [
        string: rejoin [
            "NAME:      " value/1 newline
            "ADDRESS:   " value/2 newline
            "PHONE:     " value/3 newline
            "BIRTHDAY:  " value/4
        ]
        view/new layout [
            area string
```

```
        ]
    ]
]
```

For a more detailed explanation about how to set up MYSQL, how to us the SQL language syntax, and other related topics, see http://musiclessonz.com/rebol_tutorial.html#section-27.

To use SQLite in REBOL, see http://www.dobeash.com/sqlite.html.

To use ODBC in REBOL, see http://www.rebol.com/docs/database.html.

For a useful open source SQL database system created entirely in native REBOL, see http://www.dobeash.com/rebdb.html.

The following additional database management systems ("DBMS"s) are also available for REBOL:

http://www.tretbase.com/forum/index.php
http://www.fys.ku.dk/~niclasen/nicomdb/
http://www.rebol.net/cookbook/recipes/0012.html
http://www.cs.unm.edu/~whip/
http://www.garret.ru/dybase.html
http://www.rebol.org/view-script.r?script=sql-protocol.r
http://www.rebol.org/view-script.r?script=db.r

Be sure to search rebol.org for more information and code related to databases.

9.13 Menus

One oddity about Rebol's GUI dialect is that it doesn't incorporate a native way to create standard menus. Users typically click buttons or text choices directly in REBOL GUIs to make selections. The "request-list" function and the GUI "choice" widget are short and simple substitutes which provide menu-like functionality. The menu example shown earlier in this tutorial can also be useful, but it doesn't look or operate in the typical way users expect. The popular REBOL GUI tool called RebGUI has a simple facility for creating basic menus, which can be useful.

For full blown menus with all the bells and whistles, animated icons, appropriate look-and-feel for various operating systems, and every possible display option, a module has been created to easily provide that capability: http://www.rebol.org/library/scripts/menu-system.r. Here's a minimal example demonstrating it's use:

```
do-thru http://www.rebol.org/library/scripts/menu-system.r
menu-data: [edit: item "Menu" menu [item "Item1" item "Item2"]]
simple-style: [item style action [alert item/body/text]]

view center-face layout/size [
    at 2x2 menu-bar menu menu-data menu-style simple-style
] 400x500
```

Here's a typical example that demonstrates the basic syntax for common menu layouts:

```
REBOL []

; You can download the menu-system.r script to your hard drive:

if not exists? %menu-system.r [write %menu-system.r (
        read http://www.rebol.org/library/scripts/menu-system.r)]

; If you're packaging your program into an .exe file, be sure to
; include the menu-system.r script in your package:

do %menu-system.r

; Here's how to create a menu layout:
; The "menu-data" block contains all the top level menus.
; Items in each of those menus go into separate "menu" blocks.
; Submenus are simply items with their own additional "menu" blocks.
; Use "---" for separator lines:

menu-data: [
    file: item "File" menu [item "Open" item "Save" item "Quit"]
    edit: item "Edit" menu [
        item "Item 1"
        item "Item 2" <ctrl-q>
        ---
        item "Submenu..." menu [
            item "Submenu Item 1"
            item "Submenu Item 2"
            item "Submenu Item 3" menu [
                item "Sub-Submenu Item 1"
                item "Sub-Submenu Item 2"
            ]
        ]
        ---
        item "Item 3"
    ]
```

```
    icons: item "Icons" menu [
        item "Icon Item 1" icons [help.gif stop.gif]
        item "Icon Item 2" icons [info.gif exclamation.gif]
    ]
]

; Each menu selection can now run any code you want.
; Just use the "switch" structure below:

basic-style: [item style action [
    switch item/body/text [
        ; put any code you want, in each block:
        case "Open" [
            the-file: request-file
            alert rejoin ["You opened: " the-file]
        ]
        case "Save" [
            the-file: request-file/save
            alert rejoin ["You saved to: " the-file]
        ]
        case "Quit" [
            if (request/confirm "Really Quit?") = true [quit]
        ]
        case "Item 1" [alert "Item 1 selected"]
        case "Item 2" [alert "Item 2 selected"]
        case "Item 3" [alert "Item 3 selected"]
        case "Submenu Item 1" [alert "Submenu Item 1 selected"]
        case "Submenu Item 2" [alert "Submenu Item 2 selected"]
        case "Submenu Item 3" [alert "Submenu Item 3 selected"]
        case "Sub-Submenu Item 1" [alert "Sub-Submenu Item 1 selected"]
        case "Sub-Submenu Item 2" [alert "Sub-Submenu Item 2 selected"]
        case "Icon Item 1" [alert "Icon Item 1 selected"]
        case "Icon Item 2" [alert "Icon Item 2 selected"]
    ]
]]

; The following lines need to be added to eliminate a potential problem
; closing down:

evt-close: func [face event] [either event/type = 'close [quit] [event]]
insert-event-func :evt-close

; Now put the menu in your GUI, as follows:

view center-face layout [
    size 400x500
    ; use this stock code:
    at 2x2 menu-bar menu menu-data menu-style basic-style
]
```

The demo at http://www.rebol.org/library/scripts/menu-system-demo.r shows off many more advanced features of the module.

Below is an intermediate example with explanations of the most important features. It also includes some stock code to display menus with a standard MS Windows style (OS specific appearance):

```
REBOL [Title: "Menu Example"]

; The following line imports the compressed menu module:

do http://www.rebol.org/library/scripts/menu-system.r
```

```
; Note that there are two menus in the following
; block.  The "file" menu is indented and spread
; across several lines.  The edit menu is all on
; one line.  Notice that you can place action blocks
; after each menu item, to be performed whenever the
; menu item is selected - as with the [print "You
; chose Item 1"] block below:

menu-data: [
    file: item "File"
        menu [
            new:    item "Item 1" [print "You chose Item 1"]
            open:   item "Item 2" ; icons [1.png 2.png]
            ---
            recent: item "Look In Here..."
                menu [
                    item "WIN A PRIZE!"
                    item "Try door number two"
                ]
            ---
            exit:   item <Ctrl-Q> "Exit"
          ]
    edit: item "Edit" menu [item "copy" item "paste"]
]

; Most of the style definition below is totally optional.
; It's designed to look like a native Microsoft menu.  The
; example at
; http://www.rebol.org/library/scripts/menu-system-demo.r
; contains many more examples of menu styles and options.
; The only part that's required in the example below is
; the action block in the "item style" section.  Everything
; else serves only to adjust the cosmetic appearance of the
; menu:

winxp-menu: layout-menu/style copy menu-data xp-style: [
    menu style edge [size: 1x1 color: 178.180.191 effect: none]
        color white
        spacing 2x2
        effect none
    item style
        font [name: "Tahoma" size: 11 colors: reduce [
            black black silver silver]]
        colors [none 187.183.199]
        effects none
        edge [size: 1x1 colors: reduce [none 178.180.191]
            effects: []]
        action [

            ; Change the lines below to fit your needs.
            ; You can use the action block of each item
            ; in the switch structure to run your own
            ; functions.  "item/body/text" refers to the
            ; selected menu item.  This does the exact same
            ; thing as including a code block for each item
            ; in the menu definition above (i.e., you can
            ; put the [quit] block after the "exit" item
            ; above, and it will perform the same way -
            ; just like the "[print "You chose Item 1"]"
            ; block after the "new" item above).

            switch/default item/body/text [
```

```
                "exit" [quit]
                "WIN A PRIZE!" [alert "You win!"]
                "Try door number two" [alert "Bad choice :("]
            ] [print item/body/text]  ; default thing to do
        ]
]

; The following function traps the GUI close event.  This
; must be included whenever the menu module is used, or a
; portion of the application will continue to run after being
; shut down.

evt-close: func [face event] [
    either event/type = 'close [quit] [event]
]
insert-event-func :evt-close

; And finally, here's the user interface:

window: layout [

    size  400x500

    ; The line below shows the winxp style menu:

    at 2x2 app-menu: menu-bar menu menu-data menu-style xp-style

    ; THE LINE BELOW SHOWS THE SAME MENU, WHENEVER THE BUTTON
    ; IS CLICKED:

    at 150x200 btn "Menu Button" [
        show-menu/offset window winxp-menu
        0x1 * face/size + face/offset - 1x0
    ]
]

view center-face window
```

9.14 Multi Column GUI Text Lists (Data Grids)

REBOL's built-in "text-list" GUI widget is very simple to use, but it can only display one column of data:

```
view layout [text-list data (system/locale/months)]
```

REBOL does have a built-in "list" widget for multiple column "data grid" displays, but it's a bit more complex to use than the text-list widget. Earlier in this text, Henrik Mikael Kristensen's listview module was introduced as a simple solution for creating multiple column data grid displays. It works well, but requires you to include a third party module. With a little knowledge and practice, you'll find that REBOL's built-in list widget can be very powerful and easy to use. In it's simplest form, the native list widget takes a size parameter, and 2 additional block parameters:

```
list (size) [GUI widget layout block] data [block(s) of data to display]
```

The "(size)" parameter is an XxY pair indicating the pixel size of the overall list widget. The "[GUI widget layout block]" is a layout of standard VID widgets used to display *each row of data in the grid*. The GUI elements in this block are *replicated* to display each consecutive row of data in the grid. The GUI layout block typically contains the word "across" (because these widgets are used to display *rows* of data), and it typically includes size parameters for each widget. The "data" block is made up of rows of information to be displayed in the grid. Each row of data is contained in a separate interior block:

```
view layout [
    list 220x100 [across text 100 text 100] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
```

The GUI block can contain other standard facet modifiers such as colors and spacing:

```
view layout [
    list 200x100 [across space 0 text red 100 text blue 100] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
```

The GUI block does *not* need to be comprised of only text fields. You can display the rows of data on widgets of *any type*:

```
view layout [
    list 304x100 [across space 0 button 150 button 150] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
```

```
]
```

IMPORTANT: You can make widgets in the list perform actions, just like in any other "view layout" code:

```
view layout [
    list 304x100 [
        across space 0
        button 150 [alert face/text]  ; When clicked, alert the text
        button 150 [alert face/text]  ; contained on the button's face.
    ] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
```

This means that creating *user editable* cells is very simple - just reassign the text of the clicked face, then update the display:

```
view layout [
    list 304x92 [
        across space 0
        btn 150 [face/text: request-text/default face/text show face]
        btn 150 [face/text: request-text/default face/text show face]
    ] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
```

Unintentional visual artifacts can be caused by the caret (cursor) in text widgets. To eliminate them, simply focus and unfocus the widget after updating the display:

```
view gui: layout [
    list 304x84 [
        across space 0
        text 150 [
            face/text: request-text/default face/text
            show gui focus face unfocus face
        ]
        text 150 [
            face/text: request-text/default face/text
            show gui focus face unfocus face
        ]
    ] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
```

Notice that the number of rows contained in the data block does not affect the number of rows displayed. The list always shows *as many rows as will fit in the overall pixel size of the widget* (we'll

attend to this issue later...):

```
view layout [
    list 304x100 [across space 0 button 150 button 150] data [
        ["row 1, column 1" "row 1, column 2"]
    ]
]

view layout [
    list 304x100 [across space 0 button 150 button 150] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
        ["row 5, column 1" "row 5, column 2"]
        ["row 6, column 1" "row 6, column 2"]
        ["row 7, column 1" "row 7, column 2"]
    ]
]
```

You can resize lists to stretch and fit resizable GUI windows:

```
insert-event-func [
    either event/type = 'resize [
        li/size: gui/size - 40x40
        t1/size: t2/size: as-pair (round (li/size/1 / 2)) 19
        show li unview view gui
        none
    ][event]
]
view/options gui: layout [
    li: list 220x110 [across t1: text 100 t2: text 100] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
] [resize]
```

Here's one way to stretch and/or shrink all the cells to fit inside a resizable list:

```
gui-size: 220x110  li-size: 100x19
gui-block: [
    li: list li-size [
        across
        text first (li-size / 2)  ; (1/2 the width of the list widget)
        text first (li-size / 2)
    ] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
]
insert-event-func [
    either event/type = 'resize [
        li-size: gui/size - 40x40
        unview
        view/options gui: layout gui-block [resize]
```

```
        none
    ][event]
]
view/options gui: layout gui-block [resize]
```

Of course, you can assign a label to any properly formatted data block, and display it later in a list widget:

```
x: [
    ["row 1, column 1" "row 1, column 2"]
    ["row 2, column 1" "row 2, column 2"]
    ["row 3, column 1" "row 3, column 2"]
    ["row 4, column 1" "row 4, column 2"]
]

view layout [list 220x100 [across text 100 text 100] data x]
```

That allows you to build and display multi-column lists very easily:

```
x: copy []
for i 1 12 1 [
    some-info: copy []
    append some-info (pick system/locale/months i)
    append some-info (pick system/locale/days i)
    append/only x some-info
]

view layout [list 220x240 [across text 100 text 100] data x]
```

Here's a resizable version of the script above, which has user editing enabled for the first column only:

```
x: copy []
for i 1 12 1 [
    some-info: copy []
    append some-info (pick system/locale/months i)
    append some-info (pick system/locale/days i)
    append/only x some-info
]
gui-size: 220x110  li-size: 100x19
gui-block: [
    li: list li-size [
        across
        text first (li-size / 2) [
            face/text: request-text/default face/text  ; enable user edit
            show face focus face unfocus face
        ]
        text first (li-size / 2)
    ] data x
]
insert-event-func [
    either event/type = 'resize [
        li-size: gui/size - 40x40
        unview
        view/options gui: layout gui-block [resize]
        none
    ][event]
]
```

```
view/options gui: layout gui-block [resize]
```

You can collect the entire block of user-edited data using the following code:

```
editor second second get in (list-widget-label) 'subfunc
```

For example:

```
view layout [
    the-list: list 304x100 [
        across space 0
        info 150 [face/text: request-text/default face/text show face]
        info 150 [face/text: request-text/default face/text show face]
    ] data [
        ["row 1, column 1" "row 1, column 2"]
        ["row 2, column 1" "row 2, column 2"]
        ["row 3, column 1" "row 3, column 2"]
        ["row 4, column 1" "row 4, column 2"]
    ]
    btn "Display Current Data" [
        editor second second get in the-list 'subfunc
    ]
]
```

This can be used to save and load all data in the list to files, or otherwise put to use. That makes the widget very useful for data management of all types! Take a look at this script to see one way to save and load data:

```
x: copy [
    ["row 1, column 1" "row 1, column 2"]
    ["row 2, column 1" "row 2, column 2"]
    ["row 3, column 1" "row 3, column 2"]
    ["row 4, column 1" "row 4, column 2"]
]
do qq: [view layout [
    the-list: list 304x100 [
        across space 0
        info 150 [face/text: request-text/default face/text show face]
        info 150 [face/text: request-text/default face/text show face]
    ] data x
    across
    btn "Save" [
        save to-file request-file/save
            second second get in the-list 'subfunc
        show the-list
    ]
    btn "Load" [
        x: copy load to-file request-file
        unview do qq
    ]
]]
```

9.14.1 The "Supply" Function

To enable more versatile list displays, the "data" block can be replaced with a "supply" function. "Supply" works much like a "for" loop that iterates through each row of widgets in the displayed GUI list. The "supply" function automatically creates 2 new variables which are automatically incremented

each time through the rows in the list:

1. "count": the current ROW in the list
2. "index": the current COLUMN in the current row

You can use the "count" and "index" variables to select sequential values from a block of data, using the "pick" function (in the same way as in a for loop). Typically, this is used to set the "/text" property of each widget in every row.

In the following example, every row in the list contains a single text widget. The supply function runs through each row, and sets the text property of the widget's face to be one item from the "x" block (a list of months). The loop automatically increments the "count" variable to display each of the months:

```
x: copy system/locale/months

view layout [
    list 200x300 [text 200] supply [
        face/text: pick x count
    ]
]
```

This example loops through a list of files read from the current directory:

```
x: read %.

view layout [
    list 200x400 [text 200] supply [face/text: pick x count]
]
```

You can use the "count" variable to change properties of each widget face. In this example, the color property of alternate rows is changed (one color is assigned to even counted rows, another to odd rows):

```
x: read %.

view layout [
    list 200x400 [text 200] supply [
       either even? count [face/color: white][face/color: tan]
       face/text: pick x count
    ]
]
```

You can apply actions to any widget in a list, just as you can with any other widget. Clicking on any file name in the list below will open that file in the editor:

```
x: read %.

view layout [
    list 200x400 [
        text 200 [editor to-file face/text]
    ] supply [
        face/text: pick x count
    ]
]
```

You can use the "count" variable in the supply function to build *multi-column* lists from 2 or more separate data blocks (multi column grids *are* the whole point of learning to use the list widget):

```
x: copy system/locale/months
y: copy system/locale/days

view layout [
    list 250x400 [across t1: text 50 t2: text 100 t3: text 100] supply [
        t1/text: count
        t2/text: pick x count
        t3/text: pick y count
    ]
]
```

The next example uses both the "count" and "index" variables to loop through a block with 2 columns of data. *Understanding this format is the basis for all the most complex list layouts you'll need.* Take special notice of the first line in the supply block. Once all the data from the "x" block has been looped through, if there are more rows in the list display, the index value will go past the length of the data block, and cause an error. To avoid this, you simply check if the picked value is "none", and apply a value of none to the face/text, then exit the loop:

```
x: copy []
for i 1 12 1 [
    append/only x reduce [
        pick system/locale/months i
        pick system/locale/days i
    ]
]

view layout [
    list 400x400 [across text 200 text 200] supply [
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
]
```

To help clarify the above format, here's the same example with a third row added:

```
x: copy []
for i 1 12 1 [
    append/only x reduce [
        i
        pick system/locale/months i
        pick system/locale/days i
    ]
]

view layout [
    list 250x300 [
        across
        text 50
        text 100
        text 100
    ] supply [
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
]
```

Here's an example of the directory reading example from earlier, but with two columns of data displayed (file name and size). Clicked file names still bring up the editor:

```
y: read %.
x: copy []
foreach i y [append/only x reduce [i (size? to-file i)]]

view layout [
    list 300x400 [
        across
        text 200 [editor to-file face/text]
        text 100
    ] supply [
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
]
```

The following example demonstrates how to add a slider to scroll through items in a large data block:

```
x: copy []  for i 1 397 1 [append x i]

slider-pos: 0
view layout [
    across
    the-list: list 240x400 [text 200] supply [
        count: count + slider-pos
        face/text: pick x count
    ]
    slider 16x400 [
        slider-pos: (length? x) * value
        show the-list
    ]
]
```

Here's the above slider technique applied to the earlier directory reading example:

```
x: read %.

slider-pos: 0
view layout [
    across
    the-list: list 300x400 [
        text 200 [editor to-file face/text]
    ] supply [
        count: count + slider-pos
        face/text: pick x count
    ]
    slider 16x400 [
        slider-pos: (length? x) * value
        show the-list
    ]
]
```

Here's the 2 column version of the directory reading script, with a slider attached. Be aware that clicking on any file name still reads and edits that file:

```
y: read %.
x: copy []
foreach i y [append/only x reduce [i (size? to-file i)]]
```

```
slider-pos: 0
view layout [
    across
    the-list: list 300x400 [
        across
        text 200 [editor to-file face/text]
        text 100
    ] supply [
        count: count + slider-pos
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
    slider 16x400 [
        slider-pos: (length? x) * value
        show the-list
    ]
]
```

Here's another refinement of the above script, with a third column added. The look of this display is changed by adding a line between each row (the line is drawn using a box widget), and by changing the color and font of the text:

```
y: read %.
c: 0
x: copy []
foreach i y [append/only x reduce [(c: c + 1) i (size? to-file i)]]

slider-pos: 0
view layout [
    across space 0
    the-list: list 400x400 [
        across 0 space 0x0
        text 50 purple
        text 250 bold [editor read to-file face/text]
        text 100 red italic
        return box green 400x1
    ] supply [
        count: count + slider-pos
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
    slider 16x400 [
        slider-pos: (length? x) * value
        show the-list
    ]
]
```

This example by Carl Sassenrath demonstrates a basic 4 column display:

```
REBOL []

db: [
    [ "000" "Ian Fleming" "ian" 31-Dec-2003 ]
    [ "007" "James Bond" "jb" 1-Jan-2004 ]
    [ "001" "M" "m" 2-Jan-2004 ]
    [ "ABC" "Miss Moneypenny" "missm" 3-Jan-2004]
    [ "008" "Pierce Brosnan" "pb" 4-Jan-2004 ]
    [ "009" "George Lazenby" "gl" 5-Jan-2004 ]
```

```
    [ "010" "Roger Moore" "rm" 6-Jan-2004 ]
]
sld-cnt: 0
view lst1: layout [across space 0x0
    style text text [alert form face/user-data]
    list 406x100 [
        across space 0x0 text 36 text 100 text 120 text 150
    ] supply [
        face/text: none face/user-data: none
        count: count + sld-cnt
        record: pick db count
        if not record [exit]
        n: pick [1 2 3 4] index
        face/text: pick record n
        face/user-data: record
    ]
    scl1: scroller 16x100 [
        value: to-integer value * length? db
        if value <> sld-cnt [sld-cnt: value show lst1]
    ]
]
```

The following example demonstrates how to enable users to add and remove data from a list display. Notice that after adjusting the content of your original data block and then "show"ing the list, the displayed grid is automatically updated with the new data:

```
x: copy []
for i 1 10 1 [
    append/only x reduce [form random 1000 form random 1000]
]

slider-pos: 0
view layout [
    across
    the-list: list 220x240 [across text 100 text 100] supply [
        count: count + slider-pos
        if none? q: pick x count [face/text: none exit]
        face/text: pick q index
    ]
    slider 16x240 [
        slider-pos: (length? x) * value
        show the-list
    ]
    return
    btn "Remove" [remove head x  show the-list
    ]
    btn "Add" [
        insert/only head x reduce [form random 1000 form random 1000]
        show the-list
    ]
]
```

To save user-edited contents of a GUI list created with the "supply" function, you need to use the following "set-it" code when iterating through the supply function with "count" and "index" (the "second second get in (list-widget-label) 'subfunc" trick only works for lists created using the "data" function):

```
x: copy [
    ["row 1, column 1" "row 1, column 2"]
```

```
    ["row 2, column 1" "row 2, column 2"]
    ["row 3, column 1" "row 3, column 2"]
    ["row 4, column 1" "row 4, column 2"]
]
do qq: [view gui: layout [
    the-list: list 304x100 [
        across space 0
        info 150 [face/text: request-text/default face/text show gui]
        info 150 [face/text: request-text/default face/text show gui]
    ] supply [
        either count > length? x [face/text: "" face/image: none] [
            the-list/set-it face x index count
        ]
    ]
    across
    btn "Save" [
        save to-file request-file/save x
    ]
    btn "Load" [
        x: copy load to-file request-file
        unview do qq
    ]
]]
```

Be sure to see http://www.rebol.org/view-script.r?script=list-supply-how-to.r, http://www.rebol.org/view-script.r?script=vid-usage.r, http://www.rebol.org/view-script.r?script=list-scroll-demo.r, and http://www.pat665.free.fr/gtk/rebol-view.html#sect19. for more about lists.

9.14.2 Creating Home Made Multi Column Data Grids

As it turns out, it can actually be easier and more versatile to roll your own data grids using native VID components, than it is to use the "list" widget. The following examples are based on the concept at http://www.rebol.org/view-script.r?script=presenting-text-in-columns.r. In every example, a forskip loop is used to build a visual grid of GUI widgets. The loop inserts individual text items from a data block onto each widget's face. For large lists, these example run slowly, but they can be useful for creating reasonably small displays.

The first example creates a random block of two columns of data, labeled "x". Then, a forskip loop is used to assemble a layout block of field widgets, with each row of fields containing 2 consecutive text items from the data block. That GUI block is then displayed on a pane inside a box widget, which is itself displayed inside the layout of the main window. A scroller widget is added to scroll the visible portion of the grid layout. This is accomplished by adjusting the offset of the pane which contains the whole layout of field widgets. IMPORTANT: notice that each cell in this grid is *user editable* (simply because each cell is displayed using a standard VID field widget). Also notice that the data is converted to a string with the "form" function, because fields can only display text.

```
x: copy [] for i 1 179 1 [append x reduce [i random "abcd"]]

grid: copy [across space 0]  ; the GUI block containing the grid of fields
forskip x 2 [append grid compose [field (form x/1)field (form x/2)return]]
view center-face layout [across
    g: box 400x200 with [pane: layout/tight grid pane/offset: 0x0]
    scroller [g/pane/offset/y: g/size/y - g/pane/size/y * value show g]
]
```

The next example demonstrates how to take two columns of data (blocks) and combine them into a single block that can be displayed using the layout above. First, the size of the longest block is determined using the "max" function, and a for loop is run to add consecutive items from each of the source blocks, in groups of 2, to the destination block. If either column runs out of data, blank strings are added to the rest of the destination block as column place holders.

```
x: copy []
block1: copy system/locale/months  block2: copy system/locale/days
for i 1 (max length? block1 length? block2) 1 [
    append x either g: pick block1 i [g] [""]
    append x either g: pick block2 i [g] [""]
]

grid: copy [across space 0]
forskip x 2 [append grid compose [field (form x/1)field (form x/2)return]]
view center-face layout [across
    g: box 400x200 with [pane: layout/tight grid pane/offset: 0x0]
    scroller [g/pane/offset/y: g/size/y - g/pane/size/y * value show g]
]
```

The next example demonstrates how to change the look of the grid layout, and *how to obtain a block containing all the data displayed in the grid, **including user edits***. To clarify visual separation of row data, an alternating color is assigned to each row in the grid. This is handled using a "remainder" function to check for even numbered rows. For every 4 pieces of text in the data block (every 2 displayed rows), the color is set to white. Otherwise, it's set to wheat. The most important part of this example is the line which collects all the data contained in each face of the displayed grid, and builds a block ("q") to store it.

```
x: copy [] for i 1 179 1 [append x reduce [i random "abcd"]]

grid: copy [origin 0x0 across space 0x0]
forskip x 2 [
    color: either (remainder ((index? x) - 1) 4) = 0 [white][wheat]
    append grid compose [
        field 180 (form x/1) (color) edge none
        field 180 (form x/2) (color) edge none return
    ]
]
view center-face layout [
    across space 0
    g: box 360x200 with [pane: layout grid pane/offset: 0x0]
    scroller[g/pane/offset/y: g/size/y - g/pane/size/y * value / 2 show g]
    return box 1x10 return  ; just a spacer
    btn "Get Data Block (INCLUDING USER EDITS)" [
        q: copy [] foreach face g/pane/pane [append q face/text] editor q
    ]
]
```

The next example demonstrates a number of features that really make the grid malleable and useful for entering, editing, and storing columns of data. First, the look is adjusted by changing the edges of each field style. To enable all the new features, an "update" function is created to run the line of code from the previous example which creates the "q" block of data from text displayed in every cell of the grid. In every case, the data is collected and stored in the variable "q", and the desired operation is performed on that block (adding and removing rows or data, extracting vertical columns of data, saving and loading the data to/from files on the hard drive, etc.). After the data block has been changed by an operation, the entire layout is unviewed and rebuilt using the new data (i.e., the "q" data is reassigned to the initial "x" block). The code is rerun by labeling the entire script "qq" and using the "do" function to re-evaluate it. The final button demonstrates how to collect and display a history of user edits.

```
x: copy [] for i 1 179 1 [append x reduce [i random "abcd"]]

update: does [q: copy [] foreach face g/pane/pane [append q face/text]]
do qq: [grid: copy [across space 0]
```

```
forskip x 2 [append grid compose [
    field (form x/1) 40 edge none
    field (form x/2) 260 edge [size: 1x1] return
]]
view center-face gui: layout [across space 0
    g: box 300x290 with [pane: layout/tight grid pane/offset: 0x0]
    slider 16x290 [
        g/pane/offset/y: g/size/y - g/pane/size/y * value show g
    ]
    return btn "Add" [
        row: (to-integer request-text/title "Insert at Row #:") * 2 - 1
        update insert at q row ["" ""] x: copy q unview do qq
    ]
    btn "Remove" [
        row: (to-integer request-text/title "Row # to delete:") * 2 - 1
        update remove/part (at q row) 2 x: copy q unview do qq
    ]
    btn "Col 1" [update editor extract q 2]
    btn "Col 2" [update editor extract/index q 2 2]
    btn "Save" [update save to-file request-file/save q]
    btn "Load" [x: load to-file request-file do qq]
    btn "History" [
        m: copy "ITEMS YOU'VE EDITED:^/^/" update for i 1 (length? q) 1 [
            if (to-string pick x i) <> (to-string pick q i) [
                append m rejoin [pick x i " " pick q i newline]
            ]
        ] editor m
    ]
]]
```

This final example clarifies how to add additional columns, how to use GUI widgets other than fields to display the data (text widgets, in this case), how to make the widgets perform any variety of actions, and how to get data from the grid when not every widget has text on its face. It also demonstrates some additional changes to the look of the grid.

```
x: copy [] for i 1 99 1 [append x reduce [i random 99x99 random "abcd"]]

grid: copy [origin 0x0 across space 0x0]
forskip x 3 [
    append grid compose [
        b: box 520x26 either (remainder((index? x)- 1)6)= 0 [white][beige]
        origin b/offset
        text bold 180 (form x/1)
        text 120 center blue (form x/2) [alert face/text]
        text 180 right purple (form x/3) [face/text: request-text] return
        box 520x1 green return
    ]
]
view center-face layout [
    across space 0
    g: box 520x290 with [pane: layout grid pane/offset: 0x0]
    scroller 16x290 [
        g/pane/offset/y: g/size/y - g/pane/size/y * value / 2 show g
    ]
    return box 1x10 return  ; just a spacer
    btn "Get Data Block" [
        q: copy []
        foreach face g/pane/pane [
            if face/style = 'text [append q face/text]
        ]
        editor q
```

```
    ]
]
```

These examples are useful for lists that contain ~1000 or fewer rows of data. For displays with grids larger than that, one of REBOL's other listview options should be used.

9.15 RebGUI

REBOL's VID dialect ("view layout []"), is one of the language's most attractive features. The ability to create GUI windows on multiple operating systems, with as little as 1 line of code, is practical for creating many sorts of applications. "RebGUI" is a third party GUI toolkit built on REBOL/View which replicates many of the basic components in VID, and upgrades/adds to the concept with many desirable features:

1. Modern look and feel.
2. Many powerful and useful new widgets and built-in functions: resizable tables (data grids) with automatic column sorting, trees, menus, tab and scroll panels, group boxes, tool-bars, spreadsheet, pie-chart and chat widgets, new requestors, native undo/redo, spellcheck, and translate functions (with many provided language dictionaries) for text widgets, etc.
3. Simple and elegant syntax (similar to VID).
4. Full documentation and demo code for all widgets.
5. Super simple notation to handle *automatic alignment and layout of widgets in resized windows*.
6. Config file to easily manage user settings for global UI sizes, colors, behaviors, and effects of all widgets. A built-in native requestor is also provided to adjust all these settings.
7. Automatic handling of window close events.
8. User assignable function key actions.
9. Easy, automatic handling of multiple user languages.
10. Well designed object structure to access every widget, function, and feature (and containing all necessary help information, built in).
11. The entire system compresses to just over 30k.

VID is great for building quick scripts, and many of the features found in RebGUI have been created elsewhere as VID add-ons. The menu system and listview widget described earlier in this text, for example, are more powerful than those found in RebGUI. Close events and spell checking can also be handled in other ways described earlier in this text. But for most types of applications, RebGUI provides a single, simple, integrated way to build applications with all the most commonly needed user interface features. It uses a simple, consistent language structure, and provides a clean, modern looking visual design.

RebGUI is available at http://www.dobeash.com/download.html, and several tutorials are available at http://www.dobeash.com/rebgui.html. A mirror of the required files in version 117 is available at http://musiclessonz.com/rebol_tutorial/rebgui.zip. You can also download RebGUI directly within REBOL, using the built in Viewtop. To open the Viewtop, type "desktop" into the REBOL interpreter, then click REBOL -> Demos -> RebGUI. That will download the main "rebgui.r" include file, along with the "RebDoc.r" help program, the "tour.r" demo program, and some supporting graphic images. The downloaded package will automatically run tour.r, which demonstrates many of RebGUI's features. Be sure to click the "RebDOC" button to view all the documentation necessary to use RebGUI.

All you need to use RebGUI is %rebgui.r. Copy it to an accessible folder and include the line "do %rebgui.r" (with its path, if necessary), and then you can use all the built in widgets and functions in RebGUI. A quick and dirty way to do this in Windows is to run the "request-file" function in REBOL, then click Public -> www.Dobeash.com -> RebGUI, right click rebgui.r and paste it into a folder of your choice. You can also use the following script to copy it to any folder:

```
write to-file request-file/file/title/save %rebgui.r "Save As:" {
    } read view-root/public/www.dobeash.com/RebGUI/rebgui.r
```

If you're going to use RebGUI regularly, it's a good idea to copy it directly into your main REBOL install directory (the default folder is c:\program files\rebol\view).

To build your first RebGUI interface, after running the RebGUI demo, try the following code:

```
do view-root/public/www.dobeash.com/RebGUI/rebgui.r
```

```
display "Test" [button "Click Me" [alert "Clicked"]]
do-events
```

Notice that "view layout" has been replaced with "display". This function always requires some title text. Notice also that "do-events" must be included after your RebGUI code to activate the GUI.

Once you've included %rebgui.r, you can try any of the built-in widgets and functions:

```
display "" [area] do-events  ; the "area" widget
```

Notice that the area widget above has built-in undo/redo features using [CTRL]-Z and [CTRL]-Y (REBOL's native "view layout [area]" does *not* have any undo/redo capability). A built-in *spellchecker* can also be activated using [CTRL]-S! To use the spellchecker, you need to download a dictionary from http://www.dobeash.com/RebGUI/dictionary, and unzip it into %view-root/public /www.dobeash.com/RebGUI/dictionary/

Take a look at a few of the other great widgets built into RebGUI:

```
do %rebgui.r  ; be sure to include the path, if necessary

display "Pie Chart" [pie-chart data ["VID" yellow 19 "RebGUI" red 81]]
do-events

display "Spreadsheet" [
    sheet options [size 7x7] data [a1 "very " a2 "cool" a3 "=join a1 a2"]
]
do-events

display "Chat" [
    chat data ["Nick" blue "I like RebGUI" yellow 20-sep-2009/1:00]]
do-events

display/maximize "Menu" [
    menu data [
        "File" [
             "Open" [request-file]
             "Save" [request-file]
         ]
        "About" ["Info" [alert "RebGUI is great!"]]
    ]
]
do-events
```

You can run the RebDoc.r program to see the syntax required to use any of the other RebGUI widgets, requestors and functions.

The "/close" refinement of the "display" function lets you set any action(s) you want to run when a GUI window is shut down. This can help avoid data loss from accidental window closure, and provides a way to automatically process data or run other applications when a window is closed:

```
display/close "" [area] [question "Really Close?"] do-events
```

Be sure to try the "request-ui" requestor function. It lets you easily adjust the global settings for the overall look and feel of layouts created with RebGUI on your machine. Settings are saved in the file %ui.dat, in the current working directory.

```
request-ui
```

RebGUI includes a variety of "span directives" to easily automate resizing of widgets:

```
These directives automatically set the initial size of a widget:

#L - align the right hand edge of the widget with the adjacent edge
#V - align the base edge of the widget with the adjacent edge
#O - align the left hand edge of the widget with the adjacent edge

("adjacent edge" is the edge of the adjacent widget, or the edge of
the GUI, if there is no adjacent widget.)

These directives automatically adjust the size and position of
a widget when the GUI is resized:

#H - stretch or shrink the widget to fit the window height
#W - stretch or shrink the widget to fit the window width
#X x - move the widget x number of pixels to the right
#Y y - move the widget y number of pixels downward
```

Here's an example of an area widget that stretches and shrinks to fit a resized GUI window:

```
display "" [area #HW] do-events
```

Here's a fully functional, resizable text editor application, with built-in undo/redo, spell checking, and close event handling:

```
do %rebgui.r
display/maximize/close "Text Editor" [
    menu #LHW data [
        "File" [
             "Open" [x/text: read to-file request-file show x]
             "Save" [write to-file request-file/save x/text]
         ]
    ] return
    x: area #LHW
] [question "Really Close?"] do-events
```

Now that's a lot of program for just a little code!

To really get to know RebGUI, explore its main object "ctx-rebgui":

```
? ctx-rebgui
```

The "ctx-rebgui" object is set up much like REBOL's built-in "system/view/vid" object. You can explore it using path notation. Notice that built-in help is included in the "tip" path of each widget:

```
? ctx-rebgui/widgets/tree/tip
```

Here's a quick and dirty way to view help for all the RebGUI widgets:

```
foreach i (find first ctx-rebgui/widgets 'anim) [
```

```
    do compose/deep [print rejoin[i" - "(ctx-rebgui/widgets/(i)/tip)"^/"]]
]
```

Be sure to read the main RebGUI user guide at http://www.dobeash.com/RebGUI/user-guide.html, and the cookbook at http://www.dobeash.com/RebGUI/cookbook.html. Then read through all the info in RebDoc.r, examine the code in tour.r, and get to know your way around ctx-rebgui. You'll likely find RebGUI a very handy, powerful, and easy to use toolkit.

9.16 Rebcode

REBOL provides speedy performance for most common scripting tasks. For situations where higher performance computations are required (for image processing, large looping mathematical evaluations, etc.), REBOL's "rebcode" VM acts as a sort of native cross-platform assembly language which can dramatically improve the processing speed of CPU intensive tasks that benefit from low level optimization. Rebcode uses a syntax similar to typical REBOL block/function code, and allows you to access variables used outside the Rebcode context, but it is not intended for beginner programmers. Rebcode is structured similarly to assembly language, with some additional benefits such as the ability to use built-in math functions, loops and conditional evaluations, embedded documentation, and the ability to run identically on all processors. Low level Rebcode typically improves performance speed by 10x-30x. Using Rebcode is beyond the scope of this tutorial. For more information, see http://www.rebol.com/docs/rebcode.html and http://www.rebol.net/rebcode/, and search the mailing list at rebol.org. Be sure to see the the examples at http://www.rebol.net/rebcode/docs/rebcode-demos.html:

To use rebcode, you must use a version of REBOL downloaded from http://www.rebol.net/builds/, in the section marked "Download Directories" (others don't contain the rebcode VM). Get the most recently dated version available (for Windows, at least rebview1361031.exe). Once you've downloaded a rebcode enabled interpreter, try this example:

```
do http://www.rebol.net/rebcode/demos/dot-flowers.r
```

9.17 Useful REBOL Tools

Here are some web links containing free code modules and various programs that can help you accomplish useful programmatic tasks in REBOL:

http://www.hmkdesign.dk/rebol/list-view/list-view.r - a powerful listview widget to display and manipulate formatted data in GUI applications. Perhaps the single most useful addition to the REBOL GUI language.

http://www.dobeash.com/rebdb.html - a database module written entirely in native REBOL code that lets you easily store and organize data. There's also a rich GUI library and a spell checker module that can be included in your programs: http://www.dobeash.com/rebgui.html (covered earlier)

http://www.rebol.org/cgi-bin/cgiwrap/rebol/view-script.r?script=rebzip.r - a module to compress/decompress zip formatted files.

http://www.colellachiara.com/soft/Misc/pdf-maker.r - a dialect to create pdf files directly in REBOL.

http://softinnov.org/rebol/mysql.shtml - a module to directly manipulate mysql databases within REBOL (covered earlier). A module for postgre databases is also freely available at the same site.

http://www.rebol.org/cgi-bin/cgiwrap/rebol/view-script.r?script=menu-system.r - a dialect to create all types of useful GUI menus in REBOL (covered earlier).

http://softinnov.org/rebol/uniserve.shtml - a framework to help build client-server network applications.

http://softinnov.org/cheyenne.shtml - a full featured web server written entirely in native REBOL. It enables inline, PHP-like server scripting.

http://www.rebol.net/demos/BF02D682713522AA/i-rebot.r http://www.rebol.net/demos/BF02D682713522AA/objective.r and http://www.rebol.net/demos/BF02D682713522AA/histogram.r - these examples contain a 3D engine module written entirely in native REBOL draw dialect. The module lets you easily add and manipulate 3D graphics objects in your REBOL apps (covered earlier).

http://web.archive.org/web/20030411094732/www3.sympatico.ca/gavin.mckenzie/ - a REBOL XML parser library.

http://earl.strain.at/space/rebXR - a full client/server XML-RPC implementation for REBOL (contains the parser library above). Tutorials (translated from French by Google) are available here and here.

http://box.lebeda.ws/~hmm/rswf/ - a dialect to create flash (SWF) files directly from REBOL scripts.

libwmp3.dll - the easiest way to control full featured mp3 playback in REBOL. http://www.rebol.org/view-script.r?script=mp3-player-libwmp.r demonstrates how to use it in REBOL.

http://www.rebolforces.com/articles/tui-dialect/ - a dialect to position characters on the screen in command line versions of REBOL.

http://www.rebol.net/docs/makedoc.html - converts text files into nicely formatted HTML files. *This tutorial page is written and maintained entirely with makedoc*.

http://www.rebol.org/cgi-bin/cgiwrap/rebol/view-script.r?script=layout-1.8.r - a simple visual layout designer for REBOL GUI code. Not stable enough for commercial use, but helpful for quickly laying out simple GUI designs.

http://www.crimsoneditor.com/ - a source code editor for Windows, with color highlighting especially for REBOL syntax. Quick start instructions are available at http://www.rebol.net/article/0187.html.

http://www.rebol.org - the official REBOL library - full of many additional modules and useful code fragments. The first place to look when searching for REBOL source code.

9.18 6 REBOL Flavors

This tutorial covers a version of the REBOL language interpreter called REBOL/View. REBOL/View is actually only one of several available REBOL releases. Here's a quick description of the different versions:

1. View - free to download and use, it includes language constructs used to create and manipulate graphic elements. View comes with the built-in dialect called "VID", which is a shorthand mini-language used to display common GUI widgets. View and VID dialect concepts have been integrated throughout this document. The "layout" word in a typical "view layout" GUI design actually signifies the use of VID dialect code in the enclosed block. The VID dialect is used internally by the REBOL interpreter to parse and convert simple VID code to lower level View commands, which are composed from scratch by the rudimentary display engine in REBOL. VID makes GUI creation simple, without the need to deal with graphics at a rudimentary level. But for fine control of all graphic operations, the full View language is exposed in REBOL/View, and can be mixed with VID code. View also has a built-in "draw" dialect that's used to compose and alter images on screen. Aside from graphic effects, View has built in sound, and access to the "call" function for executing command line applications. As of version 2.76, REBOL/View contains many capabilities that were previously only available in commercial versions (dll, database, encryption, SSL, and more - see below). The newest official releases of View can be download from http://rebol.com/view-platforms.html. The newest test versions are at http://www.rebol.net/builds/. Older versions are at http://rebol.com/platforms-view.html.
2. Core - a text-only version of the language that provides basic functionality. It's smaller than View (about 1/3 to 1/2 the file size), without the GUI extensions, but still fully network enabled and able to run all non-graphic REBOL code constructs. It's intended for console and server applications, such as CGI scripting, in which the GUI facilities are not needed. Core is also free and can be downloaded from http://rebol.com/platforms.html. Newest versions are at http://www.rebol.net/builds/. Older versions are at http://rebol.com/platforms-core.html.
3. View/Pro - created for professional developers, it adds encryption features, Dll access and more. Pro licenses are not free. See http://www.rebol.com/purchase.html. NOTE: STARTING IN VERSION 2.76, THESE FEATURES ARE AVAILABLE IN THE FREELY DOWNLOADABLE VERSIONS OF REBOL!
4. SDK - also intended for professionals, it adds the ability create stand-alone executables from REBOL scripts, as well as Windows registry access and more to View/Pro. SDK licenses are not free.
5. Command - another commercial solution, it adds native access to common database systems, SSL, FastCGI and other features to View/Pro. NOTE: STARTING IN VERSION 2.76, THESE FEATURES ARE AVAILABLE IN THE FREELY DOWNLOADABLE VERSIONS OF REBOL!
6. Command/SDK - combines features of SDK and Command.

Some of the functionalities provided by SDK and Command versions of REBOL have been enabled by modules, patches, and applications created by the REBOL user community. For example, mysql and postgre database access, dll access, and stand-alone executable packaging can be managed by free third party creations (search rebol.org for options). Because those solutions don't conform to official REBOL standards, and because no support for them is offered by REBOL Technologies, commercial solutions by RT are recommended for critical work.

9.19 Contexts, Bindology, Parse Wizardry, Dialects, and Other Advanced Topics

On the surface, REBOL presents itself as a simple, practical, and useful tool. Many developers tend to dismiss it because of it's simple appearance, small file size, and atypical language syntax. Those who stick with REBOL, however, eventually discover that it has some truly deep and powerful language features, not immediately apparent. Several great articles have been written which cover those topics well. If you have any interest in becoming a REBOL guru, be sure to read http://blog.revolucent.net/search/label/REBOL. The bindology article at http://www.rebol.net/wiki/Bindology, and other pages at http://www.fm.vslib.cz/~ladislav/rebol provide more understanding. Be sure to also see all the additional links in the last section of this tutorial. If your interests run deeper than simple scripting and user application development, REBOL offers unique food for thought.

10. REAL WORLD CASE STUDIES - Learning To Think In Code

At this point, you've seen most essential bits of REBOL language syntax, but you're probably still saying to yourself "that's great ... but, how do I write a complete program that does _____". To materialize any working software from an imagined design, it's obviously essential to know which language constructs are available to build pieces of a program, but "thinking in code" is just as much about organizing those bits into larger structures, knowing where to begin, and being able to break down the process into a manageable, repeatable routine. This section is intended to provide some general understanding about how to convert human design concepts into REBOL code, and how to organize your work process to approach any unique situation. A number of case studies are presented to provide insight as to how specific real life situations were satisfied.

10 A Generalized Approach Using Outlines and Pseudo Code

Software virtually never springs to life in any sort of initially finalized form. It typically *evolves* through multiple revisions, and often develops in directions originally unanticipated. There's no perfect process to achieve final designs from scratch, but certain approaches typically do prove helpful. Having a plan of attack is what gets you started writing line 1 of your code, and it's what eventually delivers a working piece of software to your user's machines. Here's a generalized routine to consider:

1. Start with a *detailed definition of what the application should do*, in human terms. You won't get anywhere in the design process until you can *describe* some form of imagined final program. *Write down* your explanation and flesh out the details of the imaginary program as much as possible. Include as much detail as possible: what should the program look like, how will the user interact with it, what sort of data will it take in, process, and return, etc.
2. Determine a list of general code and data structures related to each of the 'human-described' program goals above. Take stock of any general code patterns which relate to the operation of each imagined program component. Think about how the user will get data into and out of the program. Will the user work with a desktop GUI window, web forms that connect to a CGI script, or directly via command line interactions in the interpreter console? Consider how the *data* used in the program can be represented in code, organized, and manipulated. What types of data will be involved (text types such as strings, time values, or URLs, binary types such as images and sounds, etc.). Could the program code potentially make use of variables, block/series structures and functions, or object structures? Will the data be stored in local files, in a remote database, or just in temporary memory? Think of how the program will flow from one operation to another. How will pieces of data need to be sorted, grouped and related to one another, what types of conditional and looping operations need to be performed, what types of repeated functions need to be isolated and codified? Consider everything which is intended to *happen* in the imagined piece of software, and start thinking, "_this_ is how I could potentially accomplish _that_, in code...".
3. *Begin writing a code **outline***. It's often easiest to do this by outlining a user interface, but a flow chart of operations can be helpful too. The idea here is to begin writing a generalized code container for your working program. At this point, the outline can be filled with simple natural language PSEUDO CODE that *describes* how actual code can be organized. Starting with a user interface outline is especially helpful because it provides a starting point to actually write large code structures, and it forces you to deal with how the program will handle the input, manipulation, and output of data. Simple structures such as "view layout [button [which does this when clicked...]]", "block: [with labels and sub-blocks organized like this...], "function: (which loops through this block and saves these elements to another variable...)" can be fleshed out later with complete code.
4. Finally, move on to replacing pseudo code with actual working code. This isn't nearly as hard once you've completed the previous steps. A language dictionary/guide with cross referenced functions is very helpful at this stage. And once you're really familiar with all the available constructs in the language, all you'll likely need is an occasional syntax reminder from REBOL's built-in help. Eventually, you'll pass through the other design stages much more intuitively, and get to/through this stage very quickly.
5. As a last step, debug your working code and add/change functionality as you test and use the program.

The basic plan of attack is to always explain to yourself what the intended program should do, in human terms, and then think through how all required code structures must be organized to accomplish that goal. As an imagined program takes shape, organize your work flow using a top down approach: imagined concept -> general outline -> pseudo code description / thought process -> working code -> finished code.

The majority of code you write will flow from one user input, data definition or internal function to the next. Begin mapping out all the things that need to "happen" in the program, and the info that needs to be manipulated along the way, in order for those things to happen, from beginning to end. The process of writing an outline can be helped by thinking of how the program must begin, and what must be done before the user starts to interact with the application. Think of any data or actions that need to be defined before the program starts. Then think of what must happen to accommodate each possible interaction the user might choose. In some cases, for example, all possible actions may occur as a result of the user clicking various GUI widgets. That should elicit the thought of certain

bits of GUI code structure, and you can begin writing an outline to design a GUI interface. If you imagine an online CGI application, the user will likely work with forms on a web page. You can begin to design HTML forms, which will inevitably lead to specifying the variables that are passed to the CGI app. If your program will run as a simple command line app, the user may respond to text questions. Again, some code from the example applications in this tutorial should come to mind, and you can begin to form a coding structure that enables the general user interface and work flow.

Sometimes it's simpler to begin thinking through a development process using console interactions. It tends to be easier to develop a CGI application if you've got working console versions of various parts of the program. Whatever your conceived interface, think of all the choices the user can make at any given time, and provide a user interface component to allow for those choices. Then think of all the operations the computer must perform to react to each user choice, and describe what must happen in the code.

As you tackle each line of code, use natural language pseudo code to organize your thoughts. For example, if you imagine a button in a GUI interface doing something for your user, you don't need to immediately write the REBOL code that the button runs. Initially, just write a *description* of what you want the button to do. The same is true for functions and other chunks of code. As you flesh out your outline, ***describe*** *the language elements and coding thought you conceive to perform various actions or to represent various data structures*. The point of writing pseudo code is to keep clearly focused on the overall design of the program, at every stage of the development process. Doing that helps you to avoid getting lost in the nitty gritty syntax details of actual code. It's easy to lose sight of the big picture whenever you get involved in writing each line of code.

As you convert you pseudo code thoughts to language syntax, remember that most actions in a program occur as a result of conditional evaluations (if this happens, do this...), loops, or linear flow from one action to the next. If you're going to perform certain actions multiple times or cycle through lists of data, you'll likely need to run through some loops. If you need to work with changeable data, you'll need to define some variable words, and you'll probably need to pass them to functions to process the data. Think in those general terms first. Create a list of data and functions that are required, and put them into an order that makes the program structure build and flow from one definition, condition, loop, GUI element, action, etc., to the next.

What follows are a number of case studies that describe how I've approached various programming tasks in a productive way. Each example traces my train of thought from the organizational process through the completed code.

10.1 Case 1 - Scheduling Teachers

In my music lesson business, teachers were familiar with hand written paper schedules that looked like this:

```
Monday:

3       student1, 555-1234, parent's names, payment history, notes
3:30    student2, 555-1234, parent's names, payment history, notes
4       (gone 3-17)  student3, 555-1234, payment history, notes
4:30    student4, 555-1234, parent's names, payment history, notes
5       student5, 555-1234, parent's names, payment history, notes


Tuesday:

3       ----
3:30    ----
4       (john doe 3-18) ----
4:30    ----
5       student1, 555-1234, parent's names, payment history, notes
5:30    student2, 555-1234, parent's names, payment history, notes
6       student3, 555-1234, parent's names, payment history, notes
6:30    ----
7       student4, 555-1234, parent's names, payment history, notes
7:30    ----
8       student5, 555-1234, parent's names, payment history, notes
.
.
.
```

To run my business, I wanted to create the above schedule format on a web page, and frame it in an HTML document that had some permanent info which teachers wouldn't alter. I wanted each teacher to be able to make adjustments to their schedule without having to mess with ftp or anything having to do with the web site. I just wanted them to be able to click a desktop icon, type changes into their schedule, and have it appear on a web page. I imagined a simple application that would do those things, and came up with this basic outline of how it could work:

1. Download a teacher's current schedule text file.
2. Backup a copy of the existing schedule, just in case.
3. Edit the schedule.
4. Upload the altered schedule data back to the website.
5. Include the new schedule text in an HTML template, retaining the proper line format.
6. Confirm that the changes were made correctly and that they displayed correctly on the web page.
7. Keep the teacher interface simple and intuitive, like writing on a piece of paper.

After looking at the above outline, I just did each step above in the most direct way possible in REBOL code:

```
; first set I some initial required variables:

url: http://website.com/teacher
ftp-url: ftp://user:pass@website.com/public_html/teacher

; ... and gave the teacher some instructions:

alert {Edit your schedule, then click save and quit.
    The website will be automatically updated.}
```

```
; 1) download the file containing the schedule text:

write %schedule.txt read rejoin [url "/schedule.txt"]

; 2) create a timestamped backup on the web server:

write rejoin [ftp-url "/" now/date "_" now/time ".txt"] read %schedule.txt

; 3 and 7) edit the text:

editor %schedule.txt

; 4) save the edited text back to the web site:

write rejoin [ftp-url "/schedule.txt"] read %schedule.txt

; 6) confirm that the changes are displayed correctly:

browse url
```

To satisfy step 5 in the outline, I created a downloadable executable (".exe" file) of the above program (using XpackerX), and uploaded it to the web site. In the http://website.com/teacher folder on the web site, I created an index.cgi script containing the following code:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"

print {<a href="./scheduler.exe" target=_blank>Download Scheduler</a><br>}
print rejoin ["<pre>" read %schedule.txt "</pre>"]
```

The first HTML line creates a download link, so that the teacher can download and run his scheduler program at any remote location. The second line includes the preformatted schedule text on the web page. I can put any other HTML I want on this page, which the teacher never touches (their contact information, lesson rates, information about vacation dates, types of students they want to teach, etc.).

What could have been a very long and involved database programming task was accomplished in minutes, and was used every day for many months in the business. The free form format enabled by using a simple text file provided the opportunity to incorporate various notes, changes, and info that would otherwise be awkward to include or difficult to emphasize in a database type scheduling app. In this case, writing the pseudo code outline provided an immediate solution, and it worked out to be the best way to satisfy our needs. You'll see later how I built this basic idea into a much more complex application which runs a busy business of 25+ instructors.

10.2 Case 2 - A Simple Image Gallery CGI Program

When putting together the web site for my music lesson business, I wanted to regularly add photos of students performing at various events. At first, I just uploaded the photos individually, and added a link to the folder that contained them. As the collection grew, I wanted users to see the images more easily, without having to click on each individual file name. So, I put together a simple flash presentation that showed the images one by one. But updating that presentation required too much maintenance. What I wanted was to simply upload photos, and have them all display in a nice format on a single web page, without any required maintenance. This type of small CGI application was perfectly suited to REBOL. It only took a few minutes to write, and it now gets used every day.

For this program, here's the outline and pseudo code I worked through in my head:

1. Start by creating a simple command line script on my home computer that reads a directory listing and uses a foreach loop to run through the files and perform necessary actions.
2. Within the foreach loop, check for specified image types (extensions in each file name), and only work with those files. Add a counter to display the total number of images. To do that, use a counter variable and increment it each time through the loop.
3. In the foreach loop, wrap each image in the list in the HTML tags required to disply them on a web page. Add necessary headers to create a CGI script that runs on the web site. The script should print the HTML to the visitor's browser so they see a web page containing all the images.

Here's the code for step 1:

```
REBOL []

folder: read %.
foreach file folder [
    print file
    ; this is just a dummy action to be sure the loop is working properly
]
halt
```

For step 2, I added the counter variable, and checked for specified image types using an "if any" conditional expression:

```
REBOL []

folder: read %.
count: 0
foreach file folder [
    if any [
        find file ".jpg"
        find file ".gif"
        find file ".png"
        find file ".bmp"
    ] [
        print file
        count: count + 1
    ]
]
print rejoin [newline "Total Images: " count]
halt
```

I shortened that script a bit by using an alternate version which relies on nested foreach loops. The alternate code makes the list of potential image types easier to extend in the future:

```
REBOL []

folder: read %.
count: 0
foreach file folder [
    foreach ext [".jpg" ".gif" ".png" ".bmp"] [
        if find file ext [
            print file
            count: count + 1
        ]
    ]
]
print rejoin [newline "Total Images: " count]
halt
```

For the last step, I borrowed a line from the earlier "guitar chord diagram maker" example. It builds the HTML required to display each image on a web page. I replaced the dummy print function above with this code:

```
print rejoin [{<img src="} file {">}]
```

Finally, I added the typical CGI headers and page formatting code required to make REBOL CGI scripts perform correctly (see the previous CGI examples in this tutorial for similar patterns):

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "Photo Viewer"]
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Jam Session Photos"</TITLE></HEAD><BODY>]
print read %pageheader.html

folder: read %.
count: 0
foreach file folder [
    foreach ext [".jpg" ".gif" ".png" ".bmp"] [
        if find file ext [
            print [<BR> <CENTER>]
            print rejoin [{<img src="} file {">}]
            print [</CENTER>]
            count: count + 1
        ]
    ]
]
print [<BR>]
print rejoin ["Total Images: " count]
print read %pagefooter.html
```

I uploaded that script to the folder containing images on our web server, and updated the link to the photos on our web site. Now, we just upload new images directly to the server, and when web site visitors click the "Photos" link on our site, they instantly see a dynamically created web page full of all images currently contained in that folder.

10.3 Case 3 - Days Between Two Dates Calculator

In my business, teachers often need to figure the number of days that are between any two given dates. I can do that easily with the REBOL interpreter - just subtract any one date from another. For the unfortunate souls who don't know REBOL, I wanted to create a little GUI app that would quickly figure the calculation with some simple pointing and clicking.

This application ended up being built in stages. I started with this very simple pseudo-code idea for a script:

1. Use the "request-date" function to get a start date from the user. Assign the response to a variable.
2. Run the request-date function again to get an end date from the user. Assign that response to another variable.
3. Subtract the end-date variable from the start-date variable. Assign the result to a third variable.
4. Alert the user with the result.

That's all very straight forward. Here's the working code:

```
sd: request-date    ; get the START-DATE
ed: request-date    ; get the END-DATE
db: ed - sd         ; calculate the DAYS-BETWEEN
alert rejoin ["Days between " sd " and " ed ": " db]  ; display the result
```

That was too easy. So I decided to create a bit more of a GUI interface. Here's the pseudo-code thought process I went through:

1. Create a "view layout" window and have a separate button run each of the request-date functions (start-date and end-date).
2. Run the days-between calculation after the end-date is selected, and display the result in a text field. In order for this to happen, the numeric days-between result needs to be converted to a text string (because fields can only display text string values). Don't forget to update the displayed results with the "show" function.

Here's the code:

```
REBOL [title: "Days Between"]

view layout [
    btn "Select Start Date" [sd: request-date]
    btn "Select End Date" [
        ed: request-date
        db/text: to-string (ed - sd)
        show db
    ]
    h1 "Days Between:"
    db: field
]
```

It works, but I'd like the user to be able to see the chosen dates in text fields. Here's my pseudo-code thought process for that feature addition:

1. I'll add two more text fields to the GUI layout.
2. Whenever the user selects a new start/end date, I'll update the appropriate text field to display the selected dated. In order for that to work properly, again, I'll need to use the "to-string" function to convert the chosen date to a text value.

Here's the code I came up with to make those changes:

```
REBOL [title: "Days Between"]

view layout [

    btn "Select Start Date" [
        sd: request-date

        ; Update the start-date text field:

        sdt/text: to-string sd
        show sdt

    ]

    ; Here's the field to display the selected start-date:
    sdt: field

    btn "Select End Date" [

        ed: request-date

        ; Update the end-date text field:

        edt/text: to-string ed
        show edt

        db/text: to-string (ed - sd)
        show db

    ]

    ; Here's the field to display the chosen end-date:
    edt: field

    h1 "Days Between:"
    db: field
]
```

As it stands now, the program will crash if I select the end date before setting the start-date (because the days-between calculation tries to run without any value set for the start-date variable). In order to fix that, here's the pseudo-code thought process I went through:

1. I'll start the program by setting the "st" and "ed" variables (start-date and end-date) to an initial value of today's date ("now/date").
2. I'll display the initial start and end dates in the GUI text fields. In order for that to work properly, again I'll need to use the "to-string" function to convert the date into a text value.

Here's how the program looks when I make those changes:

```
REBOL [title: "Days Between"]

; set the initial values for start/end date:
sd: ed: now/date

view layout [
    btn "Select Start Date" [
        sd: request-date
        sdt/text: to-string sd
        show sdt
    ]
```

```
    ; show the initial start date in this field:
    sdt: field to-string sd

    btn "Select End Date" [
        ed: request-date
        edt/text: to-string ed
        show edt
        db/text: to-string (ed - sd)
        show db
    ]

    ; show the initial end date in this field:
    edt: field to-string ed

    h1 "Days Between:"
    db: field
]
```

Great, it works, but the days-between calculation still only runs when I change the end date. I'll add the days-between calculation code to the "Select Start Date" button:

```
REBOL [title: "Days Between"]

sd: ed: now/date
view layout [
    btn "Select Start Date" [
        sd: request-date
        sdt/text: to-string sd
        show sdt

        ; Run the days-between calculation, and update the display:

        db/text: to-string (ed - sd)
        show db

    ]
    sdt: field to-string sd
    btn "Select End Date" [
        ed: request-date
        edt/text: to-string ed
        show edt
        db/text: to-string (ed - sd)
        show db
    ]
    edt: field to-string ed
    h1 "Days Between:"
    db: field
]
```

As I played with the program a bit, I realized that it would be great if the user could manually enter/edit the chosen dates. Here's my thought process:

1. I'll run the days-between calculation whenever the user makes a change to the text field.
2. I'll need to stop using the "sd" and "ed" variables to perform the calculation, and instead use the *text contained in the GUI fields*, in order to be sure I'm working with any potentially edited text values.
3. Again, I'll need to pay attention to converting dates back and forth between text and date data types. Data displayed in the GUI text fields needs to be converted to a text string, using the "to-text" function, and data used to perform the days-between calculation must be converted to a date value, using the "to-date" function. REBOL automatically knows how to subtract and add

dates, but it doesn't know how to perform those types of calculations on text strings. Just use the "to-date" function to perform appropriate calculations, and it works like magic.

```
REBOL [title: "Days Between"]

sd: ed: now/date
view layout [
    btn "Select Start Date" [
        sd: request-date
        sdt/text: to-string sd
        show sdt

        ; Perform the days-between calculation using the value
        ; contained in the end-date text field (first convert
        ; that text value to a date value):

        db/text: to-string ((to-date edt/text) - sd)

        show db
    ]
    sdt: field to-string sd [

        ; Perform the days-between calculation using the values
        ; contained in the start-date and end-date text fields
        ; (first convert those text values to date values):

        db/text: to-string ((to-date edt/text) - (to-date sdt/text))

        show db
    ]
    btn "Select End Date" [
        ed: request-date
        edt/text: to-string ed
        show edt

        ; Perform the days-between calculation using the value
        ; contained in the start-date text field (first convert
        ; that text value to a date value):

        db/text: to-string (ed - (to-date sdt/text))

        show db
    ]
    edt: field to-string ed [

        ; Perform the days-between calculation using the values
        ; contained in the start-date and end-date text fields
        ; (first convert those text values to date values):

        db/text: to-string ((to-date edt/text) - (to-date sdt/text))

        show db
    ]
    h1 "Days Between:"
    db: field
]
```

Next, I realized that I wanted an additional feature. The program should also be able to figure an end date based upon a given start date and a given number of days-between. Here's the pseudo-code thought process I went through to add that feature:

1. Display an initial value of "0" days in the "db" text field (that's the number of days between the

initial start and end dates (today - today)).
2. If the user manually enters a number of days, add the given number of days to the start date, and update the end-date text field with the result (again, be sure to convert between text and date values, as in each previous example).

Simple. Here's the updated code:

```
REBOL [title: "Days Between"]

sd: ed: now/date
view layout [

    btn "Select Start Date" [
        sd: request-date
        sdt/text: to-string sd
        show sdt
        db/text: to-string ((to-date edt/text) - sd)
        show db
    ]
    sdt: field to-string sd [
        db/text: to-string ((to-date edt/text) - (to-date sdt/text))
        show db
    ]
    btn "Select End Date" [
        ed: request-date
        edt/text: to-string ed
        show edt
        db/text: to-string (ed - (to-date sdt/text))
        show db
    ]
    edt: field to-string ed [
        db/text: to-string ((to-date edt/text) - (to-date sdt/text))
        show db
    ]
    h1 "Days Between:"
    db: field "0" [

        ; Add the manually entered number of days to the start date,
        ; and update the display:

        edt/text: to-string ((to-date sdt/text) + (to-integer db/text))
        show edt

    ]
]
```

As I tested the above code, one bug became apparent. If a date is manually entered incorrectly (for example, I tried "267-Aug-2009"), the program would come to a crashing halt with an error message. To fix that, I wrapped each date calculation that involved manual text entry in an "either error? try" routine, and alerted the user with a nice message if they entered anything other than a proper date:

```
sdt: field to-string sd [
    either error? try [to-date sdt/text] [
        alert "Improper date format."
    ] [
        db/text: to-string ((to-date edt/text) - (to-date sdt/text))
        show db
    ]
]
```

```
edt: field to-string ed [
    either error? try [to-date edt/text] [
        alert "Improper date format."
    ] [
        db/text: to-string ((to-date edt/text) - (to-date sdt/text))
        show db
    ]
]
```

I also added an error check routine to the "db" text field, in case the user entered something other than a valid number of days:

```
db: field "0" [
    either error? try [to-integer db/text] [
        alert "Please enter a number."
    ] [
        edt/text: to-string (
            (to-date sdt/text) + (to-integer db/text)
        )
    ]
    show edt
]
```

At this point, every feature I can think of has been added, and all obvious bugs squashed. The evolution of this application is typical of many software case studies. Many large applications start with a basic working idea, then gradually evolve as the code is tested, user interface adjusted, features added, bugs found and eliminated, etc. That process is creative, and it can be really fun and satisfying. When writing your own applications, you have complete control to make them perform however you like :)

Here's the final code:

```
REBOL [title: "Days Between"]

sd: ed: now/date
view layout [
    btn "Select Start Date" [
        sd: request-date
        sdt/text: to-string sd
        show sdt
        db/text: to-string ((to-date edt/text) - sd)
        show db
    ]
    sdt: field to-string sd [
        either error? try [to-date sdt/text] [
            alert "Improper date format."
        ] [
            db/text: to-string ((to-date edt/text) - (to-date sdt/text))
            show db
        ]
    ]
    btn "Select End Date" [
        ed: request-date
        edt/text: to-string ed
        show edt
        db/text: to-string (ed - (to-date sdt/text))
        show db
    ]
    edt: field to-string ed [
```

```
        either error? try [to-date edt/text] [
            alert "Improper date format."
        ] [
            db/text: to-string ((to-date edt/text) - (to-date sdt/text))
            show db
        ]
    ]
    h1 "Days Between:"
    db: field "0" [
        either error? try [to-integer db/text] [
            alert "Please enter a number."
        ] [
            edt/text: to-string (
                (to-date sdt/text) + (to-integer db/text)
            )
        ]
        show edt
    ]
]
```

I packaged that script as an executable program, using XpackerX, and distributed it to all the teachers. We use it every day. (... Of course, I still just use the REBOL command line to perform my date calculations :)

10.4 Case 4 - Simple Search

It happens fairly often that I need to search for text within files on my various web site servers and on computers at my office and home. Every operating system has programs to accomplish such searches, but I'm often unhappy with the way those programs work, so I decided to create my own customized tool that operates the way I want, on every machine. This was a simple problem for which REBOL allowed me to devise a quick solution.

I started the process by thinking through the algorithm in terms of normal human activity. If I was to manually search through every file in a given folder and all it's subfolders, here's the pseudo-code that describes what I'd do:

1. Obtain a directory listing of all items in a given start folder.
2. For each item in the list, if the item is a file, read/scan it to see if it contains the given search text.
3. For each item in the list, if the item is a folder, switch into that folder and repeat steps 1-3 (I must include step 3 in step 3 itself if I want to do the same thing to every subfolder - otherwise the process would stop with 1 subfolder - very important!). When done, switch back up to the parent folder.

Step 1 is easy in REBOL code:

```
; define a starting folder:
    current-folder:  %.\
; read the directory listing:
    read current-folder
```

Step 2 isn't much more complicated:

```
; define the search text:
phrase: "the"

; for every item in the directory listing:
foreach item (read current-folder) [
    ; if the item is a file:
    if not dir? item [
        ; read/scan the file for the given phrase:
        if find (read to-file item) phrase [
            ; display the path/filename in which
            ; the search text is found:
            print rejoin [{"} phrase {" found in:  } what-dir item]
        ]
    ]
]
```

Step 3 is recursive - the actions in step 3 include executing the actions in step 3. Such recursion operations typically require creating a function that contains the actions desired, which include calling the function itself, in which those actions are contained. Here's the new code needed for step 3 - notice that the function is named "recurse" and that that "recurse" function is called within the body of that recurse function:

```
; create the function name:
recurse: func [current-folder] [
    ; for every item in the directory listing:
    foreach item (read current-folder) [
        ; if the item is a folder:
        if dir? item [
```

```
            ; change into that folder:
            change-dir item
            ;  and do all the steps in the function again:
            recurse %.\
            ; go back up to the parent directory when
            ; there are no more sub-folders:
            change-dir %..\
        ]
    ]
]
```

I put all of the code for steps 1 and 2 into that recurse function, and now it's fully operational:

```
recurse: func [current-folder] [
    foreach item (read current-folder) [
        if not dir? item [
            if find (read to-file item) phrase [
                print rejoin [{"} phrase {" found in:  } what-dir item]
            ]
        ]
    ]
    foreach item (read current-folder) [
        if dir? item [
            change-dir item
            recurse %.\
            change-dir %..\
        ]
    ]
]
```

While testing the function, I found that some of the system files could not be read. That produced a read error. I squashed that bug by adding a bit of "if error? try []" code:

```
foreach item (read current-folder) [
    if not dir? item [  if error? try [
        if find (read to-file item) phrase [
            print rejoin [{"} phrase {" found in:  } what-dir item]
        ]] [print rejoin ["error reading " item]]
    ]
]
```

To complete the program, I added a few variables to request the search text and the starting folder. I created a string variable to hold a complete text list of all files in which the search phrase was found, and I printed a little header to show that the search process had begun. When complete, the text list of files is displayed in the REBOL text editor. Here's the final version:

```
REBOL [title: "Simple Search"]

phrase: request-text/title/default "Text to Find:" "the"
start-folder: request-dir/title "Folder to Start In:"
change-dir start-folder
found-list: ""

recurse: func [current-folder] [
    foreach item (read current-folder) [
        if not dir? item [  if error? try [
            if find (read to-file item) phrase [
                print rejoin [{"} phrase {" found in:  } what-dir item]
```

```
                found-list: rejoin [found-list newline what-dir item]
            ]] [print rejoin ["error reading " item]]
        ]
    ]
    foreach item (read current-folder) [
        if dir? item [
            change-dir item
            recurse %.\
            change-dir %..\
        ]
    ]
]

print rejoin [{SEARCHING for "} phrase {" in } start-folder "...^/"]
recurse %.\
print "^/DONE^/"
editor found-list
halt
```

Next I wanted a CGI version to run on my web sites. I'll need to input my search text and starting folder using an HTML form:

```
print [<CENTER><TABLE><TR><TD>]
print [<FORM ACTION="./search.cgi">]
print ["Text to search for:" <BR>
    <INPUT TYPE="TEXT" NAME="phrase"><BR><BR>]
print ["Folder to search in:" <BR>
    <INPUT TYPE="TEXT" NAME="folder" VALUE="../yourfolder/" ><BR><BR>]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</TD></TR></TABLE></CENTER>]
```

To make this work on the web site, I'll need to include all the standard CGI headers, and decode the submitted data (this standard code format is copied from the earlier CGI examples in this tutorial):

```
#! /home/yourpath/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Search"</TITLE></HEAD><BODY>]
; print read %template_header.html

submitted: decode-cgi system/options/cgi/query-string
```

Here's the final CGI version:

```
#! /home/yourpath/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Search"</TITLE></HEAD><BODY>]
; print read %template_header.html

submitted: decode-cgi system/options/cgi/query-string

if not empty? submitted [
    phrase: submitted/2
    start-folder: to-file submitted/4
    change-dir start-folder
    found-list: ""
```

```
    recurse: func [current-folder] [
        foreach item (read current-folder) [
            if not dir? item [  if error? try [
                if find (read to-file item) phrase [
                    print rejoin [{"} phrase {" found in:  }
                        what-dir item {<BR>}]
                    found-list: rejoin [found-list newline
                        what-dir item]
                ]] [print rejoin ["error reading " item]]
            ]
        ]
        foreach item (read current-folder) [
            if dir? item [
                change-dir item
                recurse %.\
                change-dir %..\
            ]
        ]
    ]

    print rejoin [{SEARCHING for "} phrase {" in }
        start-folder {<BR><BR>}]
    recurse %.\
    print "<BR>DONE <BR>"
    ; save %found.txt found-list
    ; print read %template_footer.html
    quit
]

print [<CENTER><TABLE><TR><TD>]
print [<FORM ACTION="./search.cgi">]
print ["Text to search for:" <BR>
    <INPUT TYPE="TEXT" NAME="phrase"><BR><BR>]
print ["Folder to search in:" <BR>
    <INPUT TYPE="TEXT" NAME="folder" VALUE="../yourfolder/" ><BR><BR>]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</TD></TR></TABLE></CENTER>]
; print read %template_footer.html
```

I use this search program constantly. It took only a few minutes to write using the most basic principles and syntax patterns seen over and over again in this tutorial, and it runs on every computing device I own, including all my home and work computers, my web sites, my phone, etc.

10.5 Case 5 - A Simple Calculator Application

Next is a quick case study about how to build a small calculator application in REBOL (this is not so much a real life case study - it just seems that building a GUI calculator app is an obligatory cliche among computer programming tutorials).

I started with a pseudo-code outline of how I wanted the program's user interface to look when complete:

1. There needs to be a display area to show numerical digits as they are input, as well as the results of calculations. A simple GUI text field will work fine for that display.
2. There need to be GUI buttons to enter numerical digits and a decimal point, as well as buttons for mathematical operators, and a button to execute calculations (an "=" sign). Each of those three categories of buttons will do generally the same types of actions, so I'll create them as separate GUI styles, each with shared action blocks.

That was enough of an outline to begin writing some actual REBOL GUI code. I toyed with various window, button, and font sizes/colors until the layout looked acceptable. Here's what I came up with using the pseudo-code above:

```
view center-face layout/tight [
    size 300x350 space 0x0 across ; basic window sizing/spacing
    display: field 300x50 font-size 28 "0" return ; the display
    style butn button 100x50 [
        ; add the action code here for number buttons
    ]
    style eval button 100x50 brown font-size 13 [
        ; add the action code here for operator buttons
    ]
    butn "1"  butn "2"  butn "3"  return   ; arrange those buttons
    butn "4"  butn "5"  butn "6"  return   ; in the window
    butn "7"  butn "8"  butn "9"  return
    butn "0"  butn "."  eval "+" return
    eval "-" eval "*" eval "/" return
    button 300x50 gray font-size 16 "=" [
        ; add the action code here for "=" sign button
    ]
]
```

To turn the above display into a functioning calculator, next I needed to think about what must happen when the number buttons are clicked. Here's some pseudo-code to outline that thought process:

1. The user must be able to enter numbers that are longer than a single digit, so every time a number button is clicked, that numerical digit should be appended to the digits in the display. I'll use "rejoin" to build the display number, and then I'll set a variable to store that number, each time a new digit is clicked.
2. In the GUI code above, I started with a "0" in the display field. That'll need to be erased before any other numbers are displayed.

That's all easy enough to do in REBOL code:

```
if display/text = "0" [display/text: ""]  ; erase the displayed "0"
display/text: rejoin [display/text value]  ; build the displayed #
show display
cur-val: display/text  ; use a variable to save the displayed #
```

Now I need to think about what should happen when the operator buttons are clicked:

1. I need to assign a variable to save the number currently entered in the GUI display (that number is already saved temporarily in the "cur-val" variable above).
2. Erase the display to prepare for a new number to be entered.
3. Assign a variable to save the operator selected.

That's all very simple - in fact, it's simpler in real REBOL code than it is in pseudo-code:

```
prev-val: cur-val  ; save the displayed # in a variable
display/text: "" show display  ; erase the display
cur-eval: value  ; save the selected operator in a variable
```

Finally, I need to think about what happens when the "=" button is clicked:

1. A computation must be evaluated, using the first number entered (the "prev-val" variable above), the operator entered (the "cur-eval" variable), and the second number entered ("cur-val").
2. The display area needs to be updated to show the value of that computation.

The easiest way I could think to build the computation was to use the "rejoin" function to build a string representing the first number entered, the operator entered, and the second number entered. I could then evaluate that computation by simply using the "do" function on the built string:

```
cur-val: do rejoin [prev-val " " cur-eval " " cur-val]
display/text: cur-val
show display
```

That was all very easy. Here's the code we've got so far:

```
view center-face layout/tight [
    size 300x350 space 0x0 across
    display: field 300x50 font-size 28 "0" return
    style butn button 100x50 [
        if display/text = "0" [display/text: ""]  ; erase the "0"
        display/text: rejoin [display/text value]  ; build the #
        show display
        cur-val: display/text  ; use a variable to save the displayed #
    ]
    style eval button 100x50 brown font-size 13 [
        prev-val: cur-val
        display/text: "" show display
        cur-eval: value
    ]
    butn "1"  butn "2"  butn "3"  return
    butn "4"  butn "5"  butn "6"  return
    butn "7"  butn "8"  butn "9"  return
    butn "0"  butn "."  eval "+" return
    eval "-" eval "*" eval "/" return
    button 300x50 gray font-size 16 "=" [
        cur-val: do rejoin [prev-val " " cur-eval " " cur-val]
        display/text: cur-val
        show display
    ]
]
```

After testing the code a bit, I found a bug. Whenever the first computation is completed, any additional digits entered are appended to the total displayed from the first calculation. That happens in this line of code in the number buttons definition (the "butn" style):

```
display/text: rejoin [display/text value]
```

That problem is easily solved by setting a flag variable when the "=" button is clicked:

```
display-flag: true
```

. . . and then checking for that flag every time a number button is clicked - if the flag is set (meaning that a total is being displayed), erase the display so that a new number can be entered, and reset the flag variable:

```
if display-flag = true [display/text: "" display-flag: false]
```

That fixes the first bug. Testing the program a little more, I found another small bug. The calculator would crash with an error if the "=" sign or any of the operator buttons were clicked before numerical digits were properly entered. That was easy to fix by simply setting some default variables in the beginning of the program - that's a fundamentally good practice in any sort of programming:

```
prev-val: cur-val: 0 cur-eval: "+" display-flag: false
```

After using the program a bit more, I found another bug. If the equal sign was clicked repeatedly, it would perform calculations that weren't intended. The following line was the culprit:

```
cur-val: do rejoin [prev-val " " cur-eval " " cur-val]
```

The "cur-val" variable was updated every time the "=" button was clicked, whether or not a new number or operator was entered. To squash that bug, I just used the "display-flag" variable that was created earlier to check if a total was being displayed. I wrapped all of the action code performed when the "=" sign was clicked, into an "if" conditional, and only performed those actions if the flag had been reset (only if a total was not being displayed):

```
if display-flag <> true [ ... ]
```

Finally, there was a bug I'd had in mind from the beginning: if the user tried to divide by 0, the program would crash. To handle this situation, I added the following conditional check inside the code above:

```
if ((cur-eval = "/") and (cur-val = "0")) [
    alert "Division by 0 is not allowed." break
]
```

At this point, the program appeared to be reasonably bug free, so I decided to add an additional feature that seemed useful while testing the code. I wanted a running printout of all calculations performed, similar to paper tape on traditional printing calculators. Adding that feature was as simple as could be. At the beginning of the program I added a "print 0" line, and then added the following line changes to the "=" button:

```
prin rejoin [prev-val " " cur-eval " " cur-val " = "]
print display/text: cur-val: do rejoin [
    prev-val " " cur-eval " " cur-val
]
```

Here's the final calculator program:

```
REBOL [title: "Calculator"]

prev-val: cur-val: 0 cur-eval: "+" display-flag: false
print "0"
view center-face layout/tight [
    size 300x350 space 0x0 across
    display: field 300x50 font-size 28 "0" return
    style butn button 100x50  [
        if display-flag = true [display/text: "" display-flag: false]
        if display/text = "0" [display/text: ""]
        display/text: rejoin [display/text value]
        show display
        cur-val: display/text
    ]
    style eval button 100x50 brown font-size 13 [
        prev-val: cur-val
        display/text: "" show display
        cur-eval: value
    ]
    butn "1"  butn "2"  butn "3"  return
    butn "4"  butn "5"  butn "6"  return
    butn "7"  butn "8"  butn "9"  return
    butn "0"  butn "."  eval "+" return
    eval "-" eval "*" eval "/" return
    button 300x50 gray font-size 16 "=" [
        if display-flag <> true [
            if ((cur-eval = "/") and (cur-val = "0")) [
                alert "Division by 0 is not allowed." break
            ]
            prin rejoin [prev-val " " cur-eval " " cur-val " = "]
            print display/text: cur-val: do rejoin [
                prev-val " " cur-eval " " cur-val
            ]
            show display
            display-flag: true
        ]
    ]
]
```

10.6 Case 6 - A Backup Music Generator (Chord Accompaniment Player)

In my music lesson business, one of the things we teach is improvisation ("jam session") skills. In order for beginning students to practice, I created a simple program they could use to hear and play along with any given chord progression, at any given tempo. Building such a program with REBOL was easy. Designing an application to play pre-recorded chords from a given text list took less than a half hour.

Here's the basic outline I came up with to get started (a very basic knowledge of chord notation is required for this case study):

1. Record wave files of major, minor, dominant 7th, half diminished, diminished 7th, minor 7th, and major 7th chords on all 12 root notes (A, A#, B, C, C#, D, D#, E, F, F#, G, and G#), along with a few other commonly used chord voicings. The recordings all needed to be of short block chords, of the exact same duration and volume.
2. Compress and embed the wave files using the binary resouce embedder from earlier in this text.
3. Load each sound into memory and give each one a variable label.
4. Create a GUI with text fields for the chords to play, and the tempo. Add "play" and "stop" buttons to control the action.
5. When the "play" button is clicked, play the wave data for each chord in the given progression, using the given timing gap. There will need to be some multitasking code to enable the looping chord progressions to be stopped.
6. Add some buttons to save and load the chord progressions, along with a button to provide some help/instructions.

The first step was mechanical - no programming required. I recorded the sounds of all twelve major, minor, and dominant 7th chords using my favorite recording software and my guitar. I saved each sound as a separate wave file in 1 directory on my hard drive (I later recorded a much larger collection of chords, but this was enough to get started).

For step 2 in the outline, I used a variation of the binary resouce embedder program from earlier in this text to loop through the files in the directory:

```
REBOL []

system/options/binary-base: 64
sounds: copy []
foreach file load %./  [
    print file
    uncompressed: read/binary file
    compressed: compress to-string uncompressed
    if ((length? uncompressed) > 5000) [
        append sounds compressed
    ]
]
editor sounds
```

It provided one huge block of data containing every one of those sounds in embedded format. The output of that chord data can be seen at http://musiclessonz.com/rebol_tutorial/backup.r:

```
64#{
eJxEd2VUW833ddyIE8fdHYoVCi1S2mJ1F+ruTvvU3d3dC1RpC7S4ExwSJFgSSEhC
EmIQIvfl9//yrr32mTMz58vcNXfP2XOTEhIQx0CgRbEL4zds32dPBIFA4EnEZYFA...
```

For step 3, I placed the "load to-binary decompress" code (found earlier in this text) in front of each embedded sound file data chunk (to decompress the data and load the sound into memory for quick use). I gave each chord it's appropriate chord label (A major, Bb major, C minor, G7, etc.). In doing so, I decided to use all flat symbols for any root notes that had accidentals (i.e., F# = Gb, C# = Db,

etc. (no sharps)). Here's how the code for the A major and Bb minor chords looked:

```
a: load to-binary decompress 64#{
eJxEd2VUW833ddyIE8fdHYoVCi1S2mJ1F+ruTvvU3d3dC1RpC7S4ExwSJFgSSEhC
EmIQIvfl9//yrr32mTMz58vcNXfP2XOTEhIQx0C ...

bbm: load to-binary decompress 64#{
eJwstwVcU9//P757d9eMbQwYMWB0d4cgraCogIgdKHY38lb0rZjYXW8VOxGbDukc
3TVqCWPdv32+///j9Tj3nnvuOa9+Pc85iQtjYkr ...
```

Here's the full list of chord labels I created (the underscore symbol was a label that I gave to a silent sound that I recorded, to be used for a beat of rest). I manually labeled each of the chord data with the following labels (using my text editor's search, copy, and paste facilities, that took about ten minutes):

```
a bb b c db d eb e f gb g ab
am bbm bm cm dbm dm ebm em fm gbm gm abm
a7 bb7 b7 c7 db7 d7 eb7 e7 f7 gb7 g7 ab7
adim7 bbdim7 bdim7 cdim7 dbdim7 ddim7
ebdim7 edim7 fdim7 gbdim7 gdim7 abdim7
am7b5 bbm7b5 bm7b5 cm7b5 dbm7b5 dm7b5
ebm7b5 em7b5 fm7b5 gbm7b5 gm7b5 abm7b5
am7 bbm7 bm7 cm7 dbm7 dm7 ebm7 em7 fm7 gbm7 gm7 abm7
amaj7 bbmaj7 bmaj7 cmaj7 dbmaj7 dmaj7
ebmaj7 emaj7 fmaj7 gbmaj7 gmaj7 abmaj7
_
```

Step 4 in the outline just required building the following simple GUI. It consists of a few labels, a text area to hold the user-entered chords, a text field for the tempo, and a couple buttons to stop and start the music action. I also decided to add the buttons from step 6 - I even put all that code in here - all that was required was to save and load the contents of the text area. Simple:

```
view center-face layout [
    across
    h2 "Chords:"
    tab
    chords: area 392x300 trim {}
    return
    h2 "Delay:"
    tab
    tempo: field 50 "0.35" text "(seconds)"
    tabs 40 tab
    btn "PLAY" []
    btn "STOP" []
    btn "Save" [save to-file request-file/save chords/text]
    btn "Load" [chords/text: load read to-file request-file show chords]
    btn "HELP" [
        alert {}
    ]
]
```

Now all that's left is step 5. I started by loading the user entered list of chords into a block:

```
sounds: to-block chords/text
```

I also gave a label to the tempo, and made sure it was treated as a decimal value:

```
the-tempo: to-decimal tempo/text
```

I took the play-sound function that you've seen earlier, and used its code inside a foreach loop that played each of the sounds in the user provided list (now in the "sounds" block). Because those chord labels now refer to actual pieces of sound data that can be inserted and played directly by the sound port, this was simple:

```
wait 0
sound-port: open sound://
foreach sound sounds [
    do rejoin ["insert sound-port " reduce [sound]]
    wait sound-port
    wait the-tempo
]
```

I wrapped the above foreach loop in a forever loop, because I wanted the chord progression to repeat continuously. To stop the music, I first thought that I'd need some multitasking code, but it turns out that it was simpler than expected. All I did was create a flag variable (the word "play"), which was set to false when the GUI stop button was clicked. Inside the above foreach loop, I checked to see if the play variable had been set to false, and if so, broke out of the loop. The stop button then simply closed the sound port after setting the variable flag to false. Below is the full code for the PLAY and STOP buttons in the GUI. Simple :)

```
btn "PLAY" [
    play: true
    the-tempo: to-decimal tempo/text
    sounds: to-block chords/text
    wait 0
    sound-port: open sound://
    forever [
        foreach sound sounds [
            if play = false [break]
            do rejoin ["insert sound-port " reduce [sound]]
            wait sound-port
            wait the-tempo
        ]
        if play = false [break]
    ]
]
btn "STOP" [
    play: false
    close sound-port
]
```

To finish up the program, I added some instruction text to the alert which pops up when the help button is clicked, and I put in an example chord progression that appears in the text area by default (the chords to "Hotel California"). In testing the program, I realized that if the GUI was closed before the music was stopped, play would continue without any way to stop it. The operating system's task manager was the only way to end the music at that point. To fix that bug, I added some code to trap the close button and stop the music (set the play flag to false and close the sound port), along with a request to "really close the program?". You've seen that sort of code in several previous examples.

Here's the final program (a shortened version, WITHOUT the chord data required to play the example):

```
REBOL [title: "Chord Accompaniment Player"]
```

```
play: false
insert-event-func [
    either event/type = 'close [
        if play = true [play: false close sound-port]
        really: request "Really close the program?"
        if really = true [quit]
    ][
        event
    ]
]

{
    .
    .
    .

    all the chord data goes here

    bm: load to-binary decompress 64#{
    eJw8dgdUU0/w7k0vJNTQe++9SpUqCogIiiKgIIq9YAdU7L397NgVRAQEpEnvvfca
    CCUhQEhCIJUk9/E/7/3f2bN7z92d...
    }

    .
    .
    .

}

view center-face layout [
    across
    h2 "Chords:"
    tab
    chords: area 392x300 trim {
        bm bm bm bm
        gb7 gb7 gb7 gb7
        a a a a
        e e e e
        g g g g
        d d d d
        em em em em
        gb7 gb7 gb7 gb7
        g g g g
        d d d d
        gb7 gb7 gb7 gb7
        bm bm bm bm
        g g g g
        d d d d
        em em em em
        gb7 gb7 gb7 gb7
    }
    return
    h2 "Delay:"
    tab
    tempo: field 50 "0.35" text "(seconds)"
    tabs 40 tab
    btn "PLAY" [
        play: true
        the-tempo: to-decimal tempo/text
        sounds: to-block chords/text
        wait 0
        sound-port: open sound://
```

```
        forever [
            foreach sound sounds [
                if play = false [break]
                do rejoin ["insert sound-port " reduce [sound]]
                wait sound-port
                wait the-tempo
            ]
            if play = false [break]
        ]
    ]
    btn "STOP" [
        play: false
        close sound-port
    ]
    btn "Save" [save to-file request-file/save chords/text]
    btn "Load" [chords/text: load read to-file request-file show chords]
    btn "HELP" [
        alert {
            This program plays chord progressions.  Simply type in
            the names of the chords that you'd like played, with a
            space between each chord.  For silence, use the
            underscore ("_") character.  Set the tempo by entering a
            delay time (in fractions of second) to be paused between
            each chord.  Click the start button to play from the
            beginning, and the stop button to end.  Pressing start
            again always begins at the first chord in the
            progression.  The save and load buttons allow you to
            store to the hard drive any songs you've created.
            Chord types allowed are major triad (no chord symbol -
            just a root note), minor triad ("m"), dominant 7th
            ("7"), major 7th ("maj7"), minor 7th ("m7"), diminished
            7th ("dim7"), and half diminished 7th ("m7b5").
            *** ALL ROOT NOTES ARE LABELED WITH FLATS (NO SHARPS)
            F# = Gb, C# = Db, etc...
         }
    ]
]
```

A full, playable version, with the complete data set of embedded chords, can be found at http://musiclessonz.com/rebol_tutorial/backup.r.

Here are a few chord examples to load. All the chords:

```
a bb b c db d eb e f gb g ab
am bbm bm cm dbm dm ebm em fm gbm gm abm
a7 bb7 b7 c7 db7 d7 eb7 e7 f7 gb7 g7 ab7
adim7 bbdim7 bdim7 cdim7 dbdim7 ddim7
ebdim7 edim7 fdim7 gbdim7 gdim7 abdim7
am7b5 bbm7b5 bm7b5 cm7b5 dbm7b5 dm7b5
ebm7b5 em7b5 fm7b5 gbm7b5 gm7b5 abm7b5
am7 bbm7 bm7 cm7 dbm7 dm7 ebm7 em7 fm7 gbm7 gm7 abm7
amaj7 bbmaj7 bmaj7 cmaj7 dbmaj7 dmaj7
ebmaj7 emaj7 fmaj7 gbmaj7 gmaj7 abmaj7
_ _ _ _
```

Brown Eyed Girl:

```
g g c c g g d d7
g g c c g g d d7
g g c c g g d d7
```

```
g g c c g g d d7
g g c c g g d d7
g g c c g g d d7
c c d d  g g em em c c d d
```

10.7 Case 7 - FTP Tool

I often use REBOL's built in text editor to edit files on my web server:

```
editor ftp://user:pass@site.com/path/public_html/file.ext
```

This entire case study evolved from my use of that function. I decided to create a small script to speed up the above process. By hard coding all my FTP info directly into a GUI text field, all I have to do to edit a file on my server is run the script and change the specific file name:

```
view layout [
    p: field "ftp://user:pass@site.com/path/public_html/file.ext"
    btn "Edit" [
        editor to-url p/text
    ]
]
```

While using that script, I'd often forget the exact names of files I needed, so I decided to add a folder browsing feature to the code. Here's the thought process I went through:

1. Add a text list to the script. Instead of entering the URL of a file name in the text field, enter a folder. When I submit the URL now, the script will read the contents of that folder and display each item in the text list.
2. When I click an item in the text list, the script will join the selected file name with the given folder, and open the editor at that URL.

Here's the code - as always with REBOL, it's extremely simple:

```
view layout [
    p: field "ftp://user:pass@site.com/path/public_html/" [
        f/data: read to-url value
        show f
    ]
    f: text-list [
        editor to-url (join p/text value)
    ]
]
```

This worked well, but after using it a few times, I decided that I still wanted the option to type in a specific file name, to have it open immediately. Here's my thought process:

1. Add an "either" condition.
2. If the entered URL is a folder, do as in the previous script (I can use the dir? function to perform this check).
3. Otherwise edit the entered URL directly.

Here's the code:

```
view layout [
    p: field "ftp://user:pass@site.com/path/public_html/file.ext" [
        either dir? to-url value [
            f/data: read to-url value
            show f
        ][
            editor to-url value
        ]
```

```
    ]
    f: text-list [
        editor to-url join p/text value
    ]
]
```

I ran into some occasional problems with the dir? function, so I changed that line to read:

```
either (to-string last value) = "/" [
```

As it stands, the above script is a useful FTP editor. To create a new file, all I have to do is type its path and file name into the text field. REBOL's built in text editor automatically creates the file if it doesn't already exist. As I used this script more, I wanted to be able to navigate folders automatically (without having to type in the names of paths/files at all). Here are my thoughts:

1. If the user clicks on a folder, append the folder name to the current folder displayed in the text field, then re-read and display the contents of the new folder in the text list. This effectively changes directories.
2. To go *up* a folder (back to the previous folder), add "../" to the directory contents read from the folder currently displayed in the text field, and show that data in the text list. Also, sort the data, so "../" appears at the top of the list.
3. When the user clicks the "../", remove the last portion of the currently entered path (everything back to the prior slash symbol), update the text field, and read/display the files in that folder in the text list. For example, if the currently displayed path is "ftp://user:pass@site.com /path/public_html/folder/", remove the "folder/" portion, update the text field to "ftp://user:pass@site.com/path/public_html/", and read/display the contents of that folder.

Step 1 is easy. Just rejoin the currently displayed folder in the text field, with the value selected from the text list:

```
p/text: rejoin [p/text value]
show p
```

Step 2 is just as easy. Append "../" to the line of code that reads and displays the files in the current folder ("f/data: read to-url value"), and sort it:

```
f/data: sort append (read to-url p/text) "../"
show f
```

For step 3, we need to search for the 2nd to last "/" symbol in the currently displayed path, and remove everything after it. To do that, we'll start searching backward from the 2nd to last character (to eliminate the final "/" character in the folder, because we want the *2nd to last* "/" character). That's easy - start searching backwards from ((length? p/text) - 1). I decided to use a "for" loop starting at that index position, and decrementing by 1. Each time through the loop, pick the character at the current index position, and if it is "/", erase all characters after that index position (use the "clear" function to delete everything at (current index + 1)). Then, update the text field with the new path, read the directory contents, and display in the text list, as in steps 1 and 2 above:

```
for i ((length? p/text) - 1) 1 -1 [
    if (to-string (pick p/text i)) = "/" [
        clear at p/text (i + 1)
        show p
        f/data: sort append read to-url p/text "../"
        show f
        break  ; quit the loop once the 2nd to last "/" is found
```

```
    ]
]
```

As I looked at that code, I realized that a simpler way to do the same thing would be to use the following code. First clear the "/" at the end of the text, then clear everything after the next "/" character ("find/last" searches backward from the end):

```
clear at p/text (index? find/last p/text "/")
clear at p/text ((index? find/last p/text "/") + 1)
show p
f/data: sort append read to-url p/text "../"
show f
```

I added those changes to the current script:

```
view layout [
    p: field "ftp://user:pass@site.com/path/" [
        either dir? to-url value [
            f/data: sort append (read to-url p/text) "../"
            show f
        ][
            editor to-url value
        ]
    ]
    f: text-list [
        ; if the user selects "../", run the code from step 3:
        either (to-string value) = "../" [
            for i ((length? p/text) - 1) 1 -1 [
                if (to-string (pick p/text i)) = "/" [
                    clear at p/text (i + 1) show p
                    f/data: sort append read to-url p/text "../" show f
                    break
                ]
            ]
        ][
            ; if the user selects a folder, run code from steps 1 and 2:
            either (to-string last value) = "/" [
                p/text: rejoin [p/text value] show p
                f/data: sort append read to-url p/text "../" show f
            ][
                editor to-url rejoin [p/text value]
            ]
        ]
    ]
]
```

Now that's a useful FTP editor! We can browse through any folder and edit/save any file, just by clicking items with the mouse. I could certainly stop there, but as I used the program, more desired features kept popping up. Next, I decided to add an image viewing feature. The thought process is simple: if a selected file is an image (jpg, png, gif, or bmp), open a new GUI window, and load/display the image. Otherwise, open the file with the text editor, as before. That's easy:

```
either find [%.jpg %.png %.gif %.bmp] suffix? value [
    view/new layout [image load to-url rejoin [p/text value]]
][
    editor to-url rejoin [p/text value]
]
```

I would occasionally click a file accidentally while browsing, so I added the following line to check whether the code above should actually be run:

```
if ((request "Edit/view this file?") = true) [(do the code above)]
```

I actually have several sites that I update regularly. It would be easy to simply copy this script several times, and change the hard coded FTP information for each web site, but I wanted a more elegant solution. I decided to add a mechanism to save and load FTP info for any website, in a config file. First I created a button in the GUI to save FTP info for a site. Here's the thought process of what should happen when the button is clicked:

1. Use a text requester to ask the user for the FTP info. I'll save it in URL format, as one line, exactly the way it's typed into the GUI text field. Use the current FTP URL typed into the text field as the default text in the requester.
2. To avoid an error, stop there if the user cancels out of the requester (i.e., doesn't enter anything).
3. Use a file requester to ask the user for a text file to save the info to (default to "%ftp.cfg").
4. Add another error check to make sure the user has actually selected a file.
5. If the file doesn't exist, create it by writing the FTP URL line to a new file.
6. If the file does exist, append the FTP URL line to the existing file.
7. Alert the user that the operation is complete.

As always with REBOL, each of those steps is extremely simple:

```
btn "Save URL" [
    url: request-text/title/default "URL to save:" p/text
    if url = none [break]
    config-file: to-file request-file/file/save %/c/ftp.cfg
    if (url <> none) and (config-file <> %none) [
        if not exists? config-file [
            write/lines config-file ftp://user:pass@website.com/
        ]
        write/append/lines config-file to-url url
        alert "Saved"
    ]
]
```

Now I need a button to load saved URLs. Here's the thought process:

1. Use a file requester to have the user select a config file (default to "%ftp.cfg")
2. Use an "either" condition to check if the file exists.
3. If the file doesn't exist, notify the user that they need to first save some URLs to a config file.
4. If the file does exist, have the user select the desired FTP information from the file (one URL line from the file). An easy way to do this is with the "request-list" function. I'll load each line in the config file into a block (use a foreach loop to read and append each line in the file to a new block), and then display that list with the request-list function. When the user selects a line from the list, I'll copy the selected line of text to the GUI text list (the original text field in this program, containing the FTP information).

Again, that's all very easy to do:

```
btn "Load URL" [
    config: to-file request-file/file %/c/ftp.cfg
    either exists? config [
        if (config <> %none) [
            my-urls: copy []
            foreach item read/lines config [append my-urls item]
            if error? try [
```

```
                p/text: copy request-list "Select a URL:" my-urls
            ] [break]
        ]
    ][
        alert "First, save some URLs to that file..."
    ]
    show p
    focus p
]
```

I added a "focus" function to the end of the above button code, so that the user can just hit their [ENTER] key to connect to the server after selecting a URL from the config file. It makes sense that some users would expect to have "Load URL", "Save URL", and "Connect" buttons, so I also decided to add a separate "Connect" button to the GUI. Since clicking on the text field and clicking on the button both do the same thing, I created a "connect" function, so that code wouldn't need to be duplicated in the action block of each of those GUI widgets. In that function I added an error check, so that the program doesn't crash if the user types in incorrect FTP information:

```
connect: does [
    either (to-string last p/text) = "/" [
    if error? try [
            f/data: sort append read to-url p/text "../" show f
        ][
            alert "Not a valid FTP address, or the connection failed."
        ]
    ][
        editor to-url p/text
    ]
]
```

As I tested the code, I realized that it would be much better to increase the size of the text list and the text field, so that I could view the entire FTP URL and the listed file/folder names. 600x350 pixels works well (fits on screens with low resolution, but is big enough to see full file paths). This is how the program looks now:

```
REBOL [title: "FTP Tool"]

connect: does [
    either (to-string last p/text) = "/" [
    if error? try [
            f/data: sort append read to-url p/text "../" show f
        ][
            alert "Not a valid FTP address, or the connection failed."
        ]
    ][
        editor to-url p/text
    ]
]
view center-face layout [
    p: field 600 "ftp://user:pass@website.com/" [connect]
    across
    btn "Connect" [connect]
    btn "Load URL" [
        config: to-file request-file/file %/c/ftp.cfg
        either exists? config [
            if (config <> %none) [
                my-urls: copy []
                foreach item read/lines config [append my-urls item]
                if error? try [
```

```
                    p/text: copy request-list "Select a URL:" my-urls
                ] [break]
            ]
        ][
            alert "First, save some URLs to that file..."
        ]
        show p focus p
    ]
    btn "Save URL" [
        url: request-text/title/default "URL to save:" p/text
        if url = none [break]
        config-file: to-file request-file/file/save %/c/ftp.cfg
        if (url <> none) and (config-file <> %none) [
            if not exists? config-file [
                write/lines config-file ftp://user:pass@website.com/
            ]
            write/append/lines config-file to-url url
            alert "Saved"
        ]
    ]
    below
    f: text-list 600x350 [
        either (to-string value) = "../" [
            for i ((length? p/text) - 1) 1 -1 [
                if (to-string (pick p/text i)) = "/" [
                    clear at p/text (i + 1) show p
                    f/data: sort append read to-url p/text "../" show f
                    break
                ]
            ]
        ][
            either (to-string last value) = "/" [
                p/text: rejoin [p/text value] show p
                f/data: sort append read to-url p/text "../" show f
            ][
                if ((request "Edit/view this file?") = true) [
                    either find [%.jpg %.png %.gif %.bmp] suffix? value [
                        view/new layout [
                            image load to-url join p/text value
                        ]
                    ][
                        editor to-url rejoin [p/text value]
                    ]
                ]
            ]
        ]
    ]
]
```

As I used the program more, I recognized several additional essential features that were required. I needed to be able to: upload/download binary files, create new folders, delete, copy, rename, change permissions, and get info about existing files. I just added buttons to the existing GUI to implement each of those features. Here's my thought process and the code which does each of those things:

To get the file size and date of any selected file, I just used the "size?" and "modified?" functions built into REBOL:

```
btn "Get Info" [
    p-file: to-url rejoin [p/text f/picked]
    alert rejoin ["Size: " size? p-file " Date: " modified? p-file]
```

```
]
```

To delete an existing file, I simply use REBOL's built in "delete" function. I added a requester to confirm that the user actually wants to delete the selected file. When the operation is complete, the directory listing is updated and the user is notified with an "alert" message:

```
btn "Delete" [
    p-file: to-url request-text/title/default "File to delete:"
        join p/text f/picked
    if ((confirm: request "Are you sure?") = true) [delete p-file]
    f/data: sort append read to-url p/text "../" show f
    if confirm = true [alert "File deleted"]
]
```

Renaming a file is just as easy, using the "rename" function. Again, I just added a confirmation request and notification when complete:

```
btn "Rename" [
    new-name: to-file request-text/title/default "New File Name:"
        to-string f/picked
    if ((confirm: request "Are you sure?") = true) [
        rename (to-url join p/text f/picked) new-name
    ]
    f/data: sort append read to-url p/text "../" show f
    if confirm = true [alert "File renamed"]
]
```

Copying files on an FTP server is just as easy as copying files on your local hard drive - just read and write:

```
btn "Copy" [
    new-name: to-url request-text/title/default "New Path:"
        (join p/text f/picked)
    if ((confirm: request "Are you sure?") = true) [
        write/binary new-name read/binary to-url join p/text f/picked
    ]
    f/data: sort append read to-url p/text "../" show f
    if confirm = true [alert "File copied"]
]
```

Creating a new file is as simple as writing a file with an empty string (""):

```
btn "New File" [
    p-file: to-url request-text/title/default "New File Name:"
        join p/text "ENTER-A-FILENAME.EXT"
    if ((confirm: request "Are you sure?") = true) [
        write p-file ""
    ]
    f/data: sort append read to-url p/text "../" show f
    if confirm = true [alert "Empty file created - click to edit."]
]
```

Creating a new folder on the FTP server is also done the same way as creating a folder on your hard drive. Just use the "make-dir" function:

```
btn "New Dir" [
    make-dir x: to-url request-text/title/default "New folder:" p/text
    alert "Folder created"
    p/text: x show p
    f/data: sort append read to-url p/text "../" show f
]
```

Downloading binary files is done using the "read/binary" and "write/binary" functions. I just added some code here to find the file name (separate it from the full path of the selected file), and used a requester to present that as the suggested save-to file name:

```
btn "Download" [
    file: request-text/title/default "File:" (join p/text f/picked)
    l-file: next to-string (find/last (to-string file) "/")
    save-as: request-text/title/default "Save as..." to-string l-file
    write/binary (to-file save-as) (read/binary to-url file)
    alert "Download Complete"
]
```

Uploading is also accomplished using the "read/binary" and "write/binary" functions:

```
btn "Upload" [
    file: to-file request-file
    r-file: request-text/title/default "Save as..."
        join p/text (to-string to-relative-file file)
    write/binary (to-url r-file) (read/binary file)
    f/data: sort append read to-url p/text "../" show f
    alert "Upload Complete"
]
```

Changing file permissions (i.e., read, write, and execute on Unix/Linux servers), is done using "write/binary/allow":

```
btn "Chmod" [
    p-file: to-url request-text/default rejoin [p/text f/picked]
    chmod: to-block request-text/title/default "Permissions:"
        "read write execute"
    write/binary/allow p-file (read/binary p-file) chmod
    alert "Permissions changed"
]
```

I also created a help button to display some text held in an "instructions" variable:

```
btn-help [inform layout [backcolor white text bold as-is instructions]]
```

Here's the final code for my full featured FTP application. It's a far cry from "editor ftp://..." :) I use this program regularly (a downloadable Windows .exe is availabe at http://musiclessonz.com/rebol_tutorial/FTP_tool.exe):

```
REBOL [title: "FTP Tool"]

Instructions: {
```

```
    Enter your username, password, and FTP URL in the text field, and
    hit [ENTER].

    BE SURE TO END YOUR FTP URL PATH WITH "/".

    URLs can be saved and loaded in multiple config files for future use.

    CONFIG FILES ARE STORED AS PLAIN TEXT, SO KEEP THEM SECURE.

    Click folders to browse through any dir on your web server.  Click
    text files to open, edit and save changes back to the server.
    Click images to view.  Also upload/download any type of file,
    create new files and folders, change file names, copy and delete
    files, change permissions, etc.

}
connect: does [
    either (to-string last p/text) = "/" [
    if error? try [
            f/data: sort append read to-url p/text "../" show f
        ][
            alert "Not a valid FTP address, or the connection failed."
        ]
    ][
        editor to-url p/text
    ]
]
view center-face layout [
    p: field 600 "ftp://user:pass@website.com/" [connect]
    across
    btn "Connect" [connect]
    btn "Load URL" [
        config: to-file request-file/file %/c/ftp.cfg
        either exists? config [
            if (config <> %none) [
                my-urls: copy []
                foreach item read/lines config [append my-urls item]
                if error? try [
                    p/text: copy request-list "Select a URL:" my-urls
                ] [break]
            ]
        ][
            alert "First, save some URLs to that file..."
        ]
        show p focus p
    ]
    btn "Save URL" [
        url: request-text/title/default "URL to save:" p/text
        if url = none [break]
        config-file: to-file request-file/file/save %/c/ftp.cfg
        if (url <> none) and (config-file <> %none) [
            if not exists? config-file [
                write/lines config-file ftp://user:pass@website.com/
            ]
            write/append/lines config-file to-url url
            alert "Saved"
        ]
    ]
    below
    f: text-list 600x350 [
        either (to-string value) = "../" [
            for i ((length? p/text) - 1) 1 -1 [
                if (to-string (pick p/text i)) = "/" [
```

```
                    clear at p/text (i + 1) show p
                    f/data: sort append read to-url p/text "../" show f
                    break
                ]
            ]
        ][
            either (to-string last value) = "/" [
                p/text: rejoin [p/text value] show p
                f/data: sort append read to-url p/text "../" show f
            ][
                if ((request "Edit/view this file?") = true) [
                    either find [%.jpg %.png %.gif %.bmp] suffix? value [
                        view/new layout [
                            image load to-url join p/text value
                        ]
                    ][
                        editor to-url rejoin [p/text value]
                    ]
                ]
            ]
        ]
    ]
    across
    btn "Get Info" [
        p-file: to-url rejoin [p/text f/picked]
        alert rejoin ["Size: " size? p-file " Date: " modified? p-file]
    ]
    btn "Delete" [
        p-file: to-url request-text/title/default "File to delete:"
            join p/text f/picked
        if ((confirm: request "Are you sure?") = true) [delete p-file]
        f/data: sort append read to-url p/text "../" show f
        if confirm = true [alert "File deleted"]
    ]
    btn "Rename" [
        new-name: to-file request-text/title/default "New File Name:"
            to-string f/picked
        if ((confirm: request "Are you sure?") = true) [
            rename (to-url join p/text f/picked) new-name
        ]
        f/data: sort append read to-url p/text "../" show f
        if confirm = true [alert "File renamed"]
    ]
    btn "Copy" [
        new-name: to-url request-text/title/default "New Path:"
            (join p/text f/picked)
        if ((confirm: request "Are you sure?") = true) [
            write/binary new-name read/binary to-url join p/text f/picked
        ]
        f/data: sort append read to-url p/text "../" show f
        if confirm = true [alert "File copied"]
    ]
    btn "New File" [
        p-file: to-url request-text/title/default "New File Name:"
            join p/text "ENTER-A-FILENAME.EXT"
        if ((confirm: request "Are you sure?") = true) [
            write p-file ""
            ; editor p-file
        ]
        f/data: sort append read to-url p/text "../" show f
        if confirm = true [alert "Empty file created - click to edit."]
    ]
    btn "New Dir" [
```

```
        make-dir x: to-url request-text/title/default "New folder:" p/text
        alert "Folder created"
        p/text: x show p
        f/data: sort append read to-url p/text "../" show f
    ]
    btn "Download" [
        file: request-text/title/default "File:" (join p/text f/picked)
        l-file: next to-string (find/last (to-string file) "/")
        save-as: request-text/title/default "Save as..." to-string l-file
        write/binary (to-file save-as) (read/binary to-url file)
        alert "Download Complete"
    ]
    btn "Upload" [
        file: to-file request-file
        r-file: request-text/title/default "Save as..."
            join p/text (to-string to-relative-file file)
        write/binary (to-url r-file) (read/binary file)
        f/data: sort append read to-url p/text "../" show f
        alert "Upload Complete"
    ]
    btn "Chmod" [
        p-file: to-url request-text/default rejoin [p/text f/picked]
        chmod: to-block request-text/title/default "Permissions:"
            "read write execute"
        write/binary/allow p-file (read/binary p-file) chmod
        alert "Permissions changed"
    ]
    btn-help [inform layout[backcolor white text bold as-is instructions]]
    do [focus p]
]
```

10.8 Case 8 - Jeopardy

My fiance wanted to create a program to help train employees at work. She hoped to create a game similar to the Jeopardy TV show, which could be played with a group of employees, in order to quiz, instruct, and interact with them in an enjoyable way. Together, we devised these specifications about how the program should work:

1. Employees are organized into 2-4 teams of players who compete against each other for prizes.
2. The program displays 5 columns of boxes, each under a separate category header. The columns of boxes are divided into rows of 5, with each row displaying incremental values of $100, $200, $300, $400, and $500.
3. The host of the game operates the program and manages game play. To start off, one team chooses a category and a dollar amount to wager, and the host clicks the chosen box. When the box is clicked, an *answer* is displayed. The first player to respond gets a chance to determine the correct *question* for the given answer (i.e., "What is ____?"). The program then displays the proper question, along with some educational information related to the topic (the point of the program is to both test and teach the employees). The program then asks the host which player got the question correct or incorrect. The wagered amount is either subtracted or added to the player's score, based on correct or incorrect response, and a running total score is displayed in an area on the bottom of the screen.
4. After each question is completed, the chosen boxes are displayed as empty, and made unresponsive.
5. Winning teams continue to choose answers, and game play continues until all boxes are completed.
6. The program needs a way for the host to prepare and save the categories, answers, and questions required to play the game.

Most of the code required to create this program will revolve around the game screen design, so I started outlining the program with a GUI layout. I looked online for some images of the Jeopardy TV show, and with a little trial and error I came up with a design of buttons and boxes that satisfied the general required description:

```
REBOL [title: "Jeopardy"]

view center-face layout [
    backdrop effect [gradient 1x1 tan brown]
    style button button effect [
        gradient blue blue/2] 100x65 font [size: 30]
    style box box brown 100x35
    space 40x10
    across
    box 660x10 effect [gradient 1x0 brown black]  ; separator line
    return
    box "Category 1"
    box "Category 2"
    box "Category 3"
    box "Category 4"
    box "Category 5"
    return
    box 660x10 effect [gradient 1x0 brown black]
    return
    button "$100" []
    button "$100" []
    button "$100" []
    button "$100" []
    button "$100" []
    return
    button "$200" []
    button "$200" []
    button "$200" []
```

```
    button "$200" []
    button "$200" []
    return
    button "$300" []
    button "$300" []
    button "$300" []
    button "$300" []
    button "$300" []
    return
    button "$400" []
    button "$400" []
    button "$400" []
    button "$400" []
    button "$400" []
    return
    button "$500" []
    button "$500" []
    button "$500" []
    button "$500" []
    button "$500" []
    return
    box 660x10 effect [gradient 1x0 brown black]
    return tab
    box "Player 1:" effect [gradient 1x1 tan brown]
    player1: box white "$0" font [color: black]
    box "Player 2:" effect [gradient 1x1 tan brown]
    player2: box white "$0" font [color: black]
    return tab
    box "Player 3:" effect [gradient 1x1 tan brown]
    player3: box white "$0" font [color: black]
    box "Player 4:" effect [gradient 1x1 tan brown]
    player4: box white "$0" font [color: black]
]
```

That looks like a lot of code, but it's all just simple VID GUI layout widgets. Next I began devising the data and logic required to make the GUI operational. First, I thought about the data required to play the game, and decided to organize all the potential questions and answers into 2 separate blocks of strings:

```
answers: [
    "$100 Answer, Category 1"
    "$100 Answer, Category 2"
    "$100 Answer, Category 3"
    "$100 Answer, Category 4"
    "$100 Answer, Category 5"
    "$200 Answer, Category 1"
    "$200 Answer, Category 2"
    "$200 Answer, Category 3"
    "$200 Answer, Category 4"
    "$200 Answer, Category 5"
    "$300 Answer, Category 1"
    "$300 Answer, Category 2"
    "$300 Answer, Category 3"
    "$300 Answer, Category 4"
    "$300 Answer, Category 5"
    "$400 Answer, Category 1"
    "$400 Answer, Category 2"
    "$400 Answer, Category 3"
    "$400 Answer, Category 4"
    "$400 Answer, Category 5"
    "$500 Answer, Category 1"
```

```
    "$500 Answer, Category 2"
    "$500 Answer, Category 3"
    "$500 Answer, Category 4"
    "$500 Answer, Category 5"
]
questions: [
    "$100 Question, Category 1"
    "$100 Question, Category 2"
    "$100 Question, Category 3"
    "$100 Question, Category 4"
    "$100 Question, Category 5"
    "$200 Question, Category 1"
    "$200 Question, Category 2"
    "$200 Question, Category 3"
    "$200 Question, Category 4"
    "$200 Question, Category 5"
    "$300 Question, Category 1"
    "$300 Question, Category 2"
    "$300 Question, Category 3"
    "$300 Question, Category 4"
    "$300 Question, Category 5"
    "$400 Question, Category 1"
    "$400 Question, Category 2"
    "$400 Question, Category 3"
    "$400 Question, Category 4"
    "$400 Question, Category 5"
    "$500 Question, Category 1"
    "$500 Question, Category 2"
    "$500 Question, Category 3"
    "$500 Question, Category 4"
    "$500 Question, Category 5"
]
```

I also needed variable labels for the category headers (so that they could be edited later by the host, without having to edit any program code):

```
Category-1: "Category 1"
Category-2: "Category 2"
Category-3: "Category 3"
Category-4: "Category 4"
Category-5: "Category 5"
```

To manage game play, I realized that every button would do basically the same thing when clicked, so I created a function which would run in the action block of each button. If each button sent a unique number ID to the function, it would be easy to map each question/answer in the blocks above to individual buttons:

```
do-button: func [num] [
    ; "num" refers to the unique number parameter sent by each
    ; individual button, every time this function is executed:
    alert pick answers num
    alert pick questions num
]
```

The questions/answers in the data blocks above are arranged so that every 5 items are incremented by $100. I added the following code to assign a dollar amount to the chosen question, based on which "num" value was passed by the button (questions 1-5 = $100, 6-10 = $200, etc.):

```
if find [1 2 3 4 5] num [val: $100]
```

```
if find [6 7 8 9 10] num [val: $200]
if find [11 12 13 14 15] num [val: $300]
if find [16 17 18 19 20] num [val: $400]
if find [21 22 23 24 25] num [val: $500]
```

Now I just need to ask the host which player responded correctly or incorrectly to the alerted answer above, and assign a variable to the response ("correct"):

```
correct: request-list "Select:" [
    "Player 1 answered correctly" "Player 1 answered incorrectly"
    "Player 2 answered correctly" "Player 2 answered incorrectly"
    "Player 3 answered correctly" "Player 3 answered incorrectly"
    "Player 4 answered correctly" "Player 4 answered incorrectly"
]
```

. . . and then update the score display boxes based on the response above:

```
switch correct  [
    "Player 1 answered correctly" [
        player1/text: to-string ((to-money player1/text) + val)
        show player1
    ]
    "Player 1 answered incorrectly" [
        player1/text: to-string ((to-money player1/text) - val)
        show player1
    ]
    "Player 2 answered correctly" [
        player2/text: to-string ((to-money player2/text) + val)
        show player2
    ]
    "Player 2 answered incorrectly"[
        player2/text: to-string ((to-money player2/text) - val)
        show player2
    ]
    "Player 3 answered correctly" [
        player3/text: to-string ((to-money player3/text) + val)
        show player3
    ]
    "Player 3 answered incorrectly" [
        player3/text: to-string ((to-money player3/text) - val)
        show player3
    ]
    "Player 4 answered incorrectly"[
        player4/text: to-string ((to-money player4/text) - val)
        show player4
    ]
    "Player 4 answered correctly" [
        player4/text: to-string ((to-money player4/text) + val)
        show player4
    ]
]
```

I added that function (with a unique incremented number argument) to the action block of every button. I also added the code to erase the face and disable each button after it's been used:

```
button "$100" [face/feel: none face/text: "" do-button 1]
button "$100" [face/feel: none face/text: "" do-button 2]
button "$100" [face/feel: none face/text: "" do-button 3]
```

```
.
.
.
```

I now have a working game according to the specified outline:

```
REBOL [title: "Jeopardy"]

answers: [
    "$100 Answer, Category 1"
    "$100 Answer, Category 2"
    "$100 Answer, Category 3"
    "$100 Answer, Category 4"
    "$100 Answer, Category 5"
    "$200 Answer, Category 1"
    "$200 Answer, Category 2"
    "$200 Answer, Category 3"
    "$200 Answer, Category 4"
    "$200 Answer, Category 5"
    "$300 Answer, Category 1"
    "$300 Answer, Category 2"
    "$300 Answer, Category 3"
    "$300 Answer, Category 4"
    "$300 Answer, Category 5"
    "$400 Answer, Category 1"
    "$400 Answer, Category 2"
    "$400 Answer, Category 3"
    "$400 Answer, Category 4"
    "$400 Answer, Category 5"
    "$500 Answer, Category 1"
    "$500 Answer, Category 2"
    "$500 Answer, Category 3"
    "$500 Answer, Category 4"
    "$500 Answer, Category 5"
]
questions: [
    "$100 Question, Category 1"
    "$100 Question, Category 2"
    "$100 Question, Category 3"
    "$100 Question, Category 4"
    "$100 Question, Category 5"
    "$200 Question, Category 1"
    "$200 Question, Category 2"
    "$200 Question, Category 3"
    "$200 Question, Category 4"
    "$200 Question, Category 5"
    "$300 Question, Category 1"
    "$300 Question, Category 2"
    "$300 Question, Category 3"
    "$300 Question, Category 4"
    "$300 Question, Category 5"
    "$400 Question, Category 1"
    "$400 Question, Category 2"
    "$400 Question, Category 3"
    "$400 Question, Category 4"
    "$400 Question, Category 5"
    "$500 Question, Category 1"
    "$500 Question, Category 2"
    "$500 Question, Category 3"
    "$500 Question, Category 4"
    "$500 Question, Category 5"
```

```
]

do-button: func [num] [
    alert pick answers num
    alert pick questions num
    if find [1 2 3 4 5] num [val: $100]
    if find [6 7 8 9 10] num [val: $200]
    if find [11 12 13 14 15] num [val: $300]
    if find [16 17 18 19 20] num [val: $400]
    if find [21 22 23 24 25] num [val: $500]
    correct: request-list "Select:" ["Player 1 answered correctly"
        "Player 1 answered incorrectly" "Player 2 answered correctly"
        "Player 2 answered incorrectly" "Player 3 answered correctly"
        "Player 3 answered incorrectly" "Player 4 answered correctly"
        "Player 4 answered incorrectly"
    ]
    switch correct  [
        "Player 1 answered correctly" [
            player1/text: to-string ((to-money player1/text) + val)
            show player1
        ]
        "Player 1 answered incorrectly" [
            player1/text: to-string ((to-money player1/text) - val)
            show player1
        ]
        "Player 2 answered correctly" [
            player2/text: to-string ((to-money player2/text) + val)
            show player2
        ]
        "Player 2 answered incorrectly"[
            player2/text: to-string ((to-money player2/text) - val)
            show player2
        ]
        "Player 3 answered correctly" [
            player3/text: to-string ((to-money player3/text) + val)
            show player3
        ]
        "Player 3 answered incorrectly" [
            player3/text: to-string ((to-money player3/text) - val)
            show player3
        ]
        "Player 4 answered incorrectly"[
            player4/text: to-string ((to-money player4/text) - val)
            show player4
        ]
        "Player 4 answered correctly" [
            player4/text: to-string ((to-money player4/text) + val)
            show player4
        ]
    ]
]

view center-face layout [
    backdrop effect [gradient 1x1 tan brown]
    style button button effect [
        gradient blue blue/2] 100x65 font [size: 30]
    style box box brown 100x35
    space 40x10
    across
    box 660x10 effect [gradient 1x0 brown black]  ; separator line
    return
    box "Category 1"
    box "Category 2"
```

```
    box "Category 3"
    box "Category 4"
    box "Category 5"
    return
    box 660x10 effect [gradient 1x0 brown black]
    return
    button "$100" [face/feel: none face/text: "" do-button 1]
    button "$100" [face/feel: none face/text: "" do-button 2]
    button "$100" [face/feel: none face/text: "" do-button 3]
    button "$100" [face/feel: none face/text: "" do-button 4]
    button "$100" [face/feel: none face/text: "" do-button 5]
    return
    button "$200" [face/feel: none face/text: "" do-button 6]
    button "$200" [face/feel: none face/text: "" do-button 7]
    button "$200" [face/feel: none face/text: "" do-button 8]
    button "$200" [face/feel: none face/text: "" do-button 9]
    button "$200" [face/feel: none face/text: "" do-button 10]
    return
    button "$300" [face/feel: none face/text: "" do-button 11]
    button "$300" [face/feel: none face/text: "" do-button 12]
    button "$300" [face/feel: none face/text: "" do-button 13]
    button "$300" [face/feel: none face/text: "" do-button 14]
    button "$300" [face/feel: none face/text: "" do-button 15]
    return
    button "$400" [face/feel: none face/text: "" do-button 16]
    button "$400" [face/feel: none face/text: "" do-button 17]
    button "$400" [face/feel: none face/text: "" do-button 18]
    button "$400" [face/feel: none face/text: "" do-button 19]
    button "$400" [face/feel: none face/text: "" do-button 20]
    return
    button "$500" [face/feel: none face/text: "" do-button 21]
    button "$500" [face/feel: none face/text: "" do-button 22]
    button "$500" [face/feel: none face/text: "" do-button 23]
    button "$500" [face/feel: none face/text: "" do-button 24]
    button "$500" [face/feel: none face/text: "" do-button 25]
    return
    box 660x10 effect [gradient 1x0 brown black]
    return tab
    box "Player 1:" effect [gradient 1x1 tan brown]
    player1: box white "$0" font [color: black]
    box "Player 2:" effect [gradient 1x1 tan brown]
    player2: box white "$0" font [color: black]
    return tab
    box "Player 3:" effect [gradient 1x1 tan brown]
    player3: box white "$0" font [color: black]
    box "Player 4:" effect [gradient 1x1 tan brown]
    player4: box white "$0" font [color: black]
]
```

After playing the game a bit, there were no bugs, but I realized that it could use a few additional features. First, I used the binary resouce embedder from earlier in this tutorial to create an image header of a photo I found online from the Jeopardy TV show:

```
header: load to-binary decompress 64#{
eJyVj3s804v/xz/bzGyFucRGohgm1ziuuU7CNGxWuTaxbBpyLVJUpItLcyfXNIZI
jkuJboqFJsNodG9uuXRyya18ncfj93t8//4+3/+9X6/H6/1+bY1ufQKQzg5ODgAI
BAIo2wNsjQE4QFQEKgoVERWFisJgomIIJAIBhyNkJSR3IuVlUSh5WTk5BSWMsoKi
6m45ORVtFVV1DawWFq2so6+jqY/RxGr+GwKCwWAIMYQMAiGjqSCnoPk/s/UMkBID
dAEmBLQPAEuBIFKgrZeAIgBAQP8C/D/bT4qIbS+3xUNIAASGgiFQsNhOKPRfEQSG
ANtlpKT3wmQMxGT32br7X5JHHTAKiGCW75JTNbQjkk6GZzYOqGl63Hl++UrSA952
jAIIAEH/e2AbMACCbMdsa5pSABgMgkBEwJD/c4DAAERKRNpAZq8tdJ+7PzHc7lL5
```

```
gy0BsHPbAJaCSAHWwNFKVxP98V4tOkWdfD7FEjacdjSkarU//C/n4IDH8tIsGPlr
F8jSOHVpqLEgNs++1taBHGuekzY4eM/ojEII4R/1Q+ZIEH2QWTpkVWn+ntRBthgZ
kympJr8jt/eRTxvTS1RNGD3meRMcm4kAZ2CAnTULVlJflfeI8BVvJWeJaNZitC7j
HSzwe21RvYdfbQEZcgNoufW05RxBPLziaGaLDnMiIu5cgjXhKavuH/3ewXQVtg6Q
BLgRCGfHVnFmn6LPF/fJDI1/TejRG5nB2X8vi0llhhEm3Fb/XnK+KeG/sKxkl6Py
E3ZteGrG4qqpYSazc72YsFrY6n0ht0oo2EEs4dhdJjJvOThxbRVx3ruRD921c4ct
XXp89nPCL1Ikh1ZwlkpvLRz53sKF8WfDpRW0V7dePB2671umDPyzp0PJxckwXbHc
pTnSOjTC54hLNXHdWIVdxdi9pDKhqNWOrCdNu3LPTh8e6tlxg9pQZw/KFHWY2OGM
Z33g/Y140axOK5pkwP2FSiCj0RhPPofjhXBtt52xdJjhjMZC9IMH+GNXzhlmhij6
fGjVfqN1lIHPvdV686aDXaIb6tn6gH1kYn1bpyM5Fi6c6cIsFn39lXJXLybMmRTv
soo4Nv7eqf/Byy2ANn2FByPXt1Xt8KZQEjhR08GMzINfIqIfBVT7HI4YKYlv/MRP
bv94k1JMSbSZATZOlnwEa64bdZBL/dZ6iEmlv1FwwpnNoCoplqla0drmB70Sd/H1
sZ9ijEDcfSc3TrdMbtAs3ZOg0+s96CofOOEnUod2Xqo2TylA/5xW1LtJzLNtp8vS
1UwD/YqprdJffrIwcdFhwvmr+CKmQDfT+P2R2AvojAzZ4RItnkwq00z74hIujcx1
Mzh42OioCXKvxeU3Ze4LhSJNyZYt72K7XrxlYf0p1TZX2MXlrulNBYgLUc2umSfJ
9fPvBnVHGUEaCRImNyxX74XvEyg79i/XIB/bfxsP1DCbf7O03rV/mvFg3u344qJ6
8IDbp3PMP47rJyasP1ii1wgX3yT4Zzh8dPztIlE1r3yH+6TqFHKa0ULlS3gTzBPE
oj4lDr4sEL9lrb7w2lG5jL3+g8+L+V5QmpvPdnGOED5HQNzNxvqubX52yVriGraq
tO6Jrj6r7imhjdzFQqA3yqLzMDmjNeevW4r1uGeg03qjg5oeh8z8EE0fVZ7+TelY
e3W27vblk7f1nIP9tWUx+PCfl50zdzXALUJVIesptLVwlles2qRi0C0KawvonDkS
0zL7bfgZvd+C5RxyaI+NmlSmpC36BtZYzzsVXrEgGDPrIzSG/uVbdKpq2DcrlPyi
N42GS+7PFLRyYffvkrYAZeof6SN31Ir1RJfPpgUQok53lbrErr+803Wn8lLXOyzc
bNkDgxT0n0PxnGhFDEeBBEs3nirrFOlQfMk1ZaKTK+lOZkzuhnKYwwE1xeMToRUa
m1188t34vrAtIIR9quLENPsjKebLtLCelN50PXG+jtoA74lKjV7Wr3Oc3Az4bqHH
fC+6atqPZVOY5d5DeF5My+8XnELv+nu97Rt08WoK2HySEbXy6rSNWuimAyaJS7A6
SX0ou0A3F0RV4b2KuCabPbAT8W5z54Vty8Izbu7qnrLNNLxwfLHB5liKIqWUNJ90
UfHge9e0PKqdZ/zrQxifKoPGRGGVylrq3QXxlbRaHMtvl1ezceO38aWZYhJPuMJH
ejSw/FUCKXPtlhuR899Au2LnD/fl/tDL75bbTwxWxWELbRTCUw+KEEO4+x4f+yK7
3NIWrBDsOyJweXaOzW5N2+4XZNVQ8GIlt3u37tUzZQKfQtWQdgLrxKIsAlpsf6tD
o/ZAtHwp++FX7iNdP1t88Jk0P6Rhaytfjl1r5fos47hqjFXgyu3S1PEfY9U/P058
QWn5uLJ7wslybYFUsvopb0qbc/Nnm7CRPWdKT7sQuDHZpfn5uRG+T86/YcEu3Hrt
qNEbH/QR/R0a0GSdJJ3lFmDdk2+Vrw2R/IrOT3wZ3zliXC5Qzz5s5/XnCsqU3Ryv
RyQmaybczOGgx6kGS0+DCivDV/LMjJpNtDiBbQ1wOFQtcMMkZZ21r+++T2P02ZpG
riura80pOBzlxXNgZhgO0q4p3T6ePqo/9ojap3RR5kcx41Jp9ustIK5ctUlT/Zsb
SqOttfW3g+LDtANDVTJz3ZW78YyWXH4z+5MatFYdaTpNZtaOeCzr0gsqN9pkqlnB
aJE5SAUx9MS4hQRy8gt+7QTuz+hSoPD+StBsxmHcpWWzji2A+PlddStmXqvTld7D
It5HXH6KH6UkVzP4LFPesu165Bzn7PU8M+Eeca0dcPOOR8hk2tBMbjM1GnG6Th4P
bTaewuxw9UHqx6WXdQZ1QgV1Wfvt0esmCRs35On1p7W1RHe+rRTwm3eeS/YJwTX1
7seY44NCcNbGVr4UV7kQr2W7n+w+J+LEqePRR6qbW9KcsOJ3lRuuNGmaedb2fnia
6r+SRnLzuwMnTD0nWOqHS+iGR0zPl9NCc3fVwrTvo/6EMaQHown3/SQZuBn+85z0
A5Qn0pwjxFWWaw5bnzpBTHGiBzh3eZuO3bPe4ffgx8rpjoETl/GxCwpXrQ0rUNzI
/GzFqw/+umAS0bYoPoTMSieKOYANUqVjVOt/dJl2R8VWFsLYeUcT6hG/LbnCQaWu
nYh71cfoGmhOulEmjHajoedAYNc79Jq4fcZ13veLs3U0nW8efhb3IANWlUaGEXz6
SOqrnr5OXQ9lGobK/1kkW0UBoaYMwLUk0i5sSk1kr21+Yb53rXgFBIOdMpShR70h
l/ejIBs3xoZzsiML7GcHNCtryNM5U3u1WxKJDfcCM+MSQKBzcHP5l27zTafCJWtB
ODP3kcH3oNnHd6pC/nY/mV99Rv1tdJ6ClirctOvPIGhOXqMuuK+OWNgib6At5uzn
hDUjZBUh45SPH50uEPDDVJsQc3AvvfEGL+magCbq2xiep1ZX3aYTbenXi9+g4dfh
73l5xFvLRDrUvMFD1vLgr+e/bK6GIV6m/XJ7pBB8iLQo/iu63OmYtppIog038vOe
M0qCnEPH7Ds4AirpYAGrL/tc0y4lDdT3jHiSLyb1w0K38rcmQRJwaRHrX9ZPT6Y/
kzCWO9z6mFnaBuOHBH/rVhqrMcQXkE0JBz7nD3ziZVdpWE35yFO9Tq4Fz8T5lWjk
U1qQiR0I1OBDbpzMpLjyzuVxCgmbtOHcvflM3HK5lPbPNYf1oj2/U72/zl+UOsEj
SsbQVq/tcZMeKOMHTdJIxKZ3KaxBN7veuhaPRHBWpofxGGz2ksnNHPPlfguPGhfj
r/NsDNaRd5VnDHf8s5byWOa6TG5atR7hkLEfVbytUc6X6PR6eHpj4CyLau5BBrL2
d1wP2QJsHD7ouDj3hiskzdk2zN5mlIep3NXEgk5zuwNas4+s5hwvI9ck90b4FtBD
iSSWDqbMQ2IS6FIsEbJ9RxI+Y4V7c3HQSmqh4l4Vmbb0GTMNnrRjD8cyUiouJjHT
kyuWUsMHnnAXTzE3nkbF6X/2ezD1aHCjwNdz691/ABEeZGXYCwAA
}
```

Adding it to the top of the GUI was this easy. I didn't edit the image to fit, but instead just used REBOL's built in ability to simply resize the image:

```
image header 660x40
```

Next, I decided to add an option to resize the entire GUI, so that the program could run on computers with varied screen resolutions. I looked through all the GUI code and realized that all the widget sizes were divisible by a factor of 5, I stored that as a variable word "sizer":

```
sizer: 5
```

Then, anywhere in the GUI where there was a sizing pair, I added a little math calculation to dynamically specify the size. The image size above (660x40), for example, was written as follows:

```
image header (to-pair rejoin [(132 * sizer) "x" (8 * sizer)])
```

Tabs and pad sizes were set like this:

```
tabs (sizer * 20)
pad (sizer * 2)
```

With those sizing calculations added to every widget, all the host needed to do was change the one sizer variable, and the whole pixel size of the game would be adjusted.

Next, I realized that the host could potentially make mistakes in game play (I did while testing the program), so I wanted a way to manually adjust game scores. I added the following code to the action block of the score boxes, which allows the host to click on the box, and enter a new value to be shown on the box's face:

```
player1: box white "$0" font [color: black size: (sizer * 4)] [
    face/text: request-text/title/default "Enter Score:" face/text
]
```

According to the last item in our specification outline, all I need now is a way for the host to edit and save game data. I started by assigning a variable to all the code that contained variables which the host should be able to edit. I wanted this code to be writable to the hard drive, and "do"able, so I stored it as a text string and included a REBOL header:

```
config: {

    REBOL []

    ;_____________________________________________________________


    sizer: 5

    Category-1: "Category 1"
    Category-2: "Category 2"
    Category-3: "Category 3"
    Category-4: "Category 4"
    Category-5: "Category 5"

    answers: [
        "$100 Answer, Category 1"
        "$100 Answer, Category 2"
```

```
        "$100 Answer, Category 3"
        "$100 Answer, Category 4"
        "$100 Answer, Category 5"
        "$200 Answer, Category 1"
        "$200 Answer, Category 2"
        "$200 Answer, Category 3"
        "$200 Answer, Category 4"
        "$200 Answer, Category 5"
        "$300 Answer, Category 1"
        "$300 Answer, Category 2"
        "$300 Answer, Category 3"
        "$300 Answer, Category 4"
        "$300 Answer, Category 5"
        "$400 Answer, Category 1"
        "$400 Answer, Category 2"
        "$400 Answer, Category 3"
        "$400 Answer, Category 4"
        "$400 Answer, Category 5"
        "$500 Answer, Category 1"
        "$500 Answer, Category 2"
        "$500 Answer, Category 3"
        "$500 Answer, Category 4"
        "$500 Answer, Category 5"
    ]
    questions: [
        "$100 Question, Category 1"
        "$100 Question, Category 2"
        "$100 Question, Category 3"
        "$100 Question, Category 4"
        "$100 Question, Category 5"
        "$200 Question, Category 1"
        "$200 Question, Category 2"
        "$200 Question, Category 3"
        "$200 Question, Category 4"
        "$200 Question, Category 5"
        "$300 Question, Category 1"
        "$300 Question, Category 2"
        "$300 Question, Category 3"
        "$300 Question, Category 4"
        "$300 Question, Category 5"
        "$400 Question, Category 1"
        "$400 Question, Category 2"
        "$400 Question, Category 3"
        "$400 Question, Category 4"
        "$400 Question, Category 5"
        "$500 Question, Category 1"
        "$500 Question, Category 2"
        "$500 Question, Category 3"
        "$500 Question, Category 4"
        "$500 Question, Category 5"
    ]

    ;_____________________________________________________________________

}
```

To use that code in normal game play, I just executed the code contained in the "config" variable:

```
do config
```

Next, I outlined some pseudo code describing each step the host might go through to edit the config

data:

1. Warn the host that these steps will erase the current data and end the current game. A simple requester and if condition will serve this purpose. Break out of the current block of code if they choose to continue the game.
2. Before going through the process of editing/saving, request if the user would simply like to load a previously edited configuration file. If so, just run ("do") the chosen config file, close the current GUI, and rerun the GUI using the new config data.
3. If the user hasn't chosen either of the previous options, give them some directions about how to edit the file, save the default config code to a file, and open it in the built in editor. After the editor has been closed, request a config file to load, then close the current GUI and rerun it using the newly chosen config data.

Here's the code I created to satisfy each of the above steps:

```
; step 1

contin: request/confirm {
    This will end the current game.  Continue?}
if contin = false [break]

; step 2

loadoredit:  request/confirm "Load previously edited config file?"
if loadoredit = true [
    do to-file request-file/title/file {
        Choose config file to use:} "File" %default_config.txt
    unview
    view center-face layout gui
    break  ; needed so that step 3 doesn't run
]

; step 3

alert {Edit carefully, maintaining all quotation marks.
    You can open a previously saved file if needed.
    When done, click SAVE-AS and then QUIT.
    Be sure choose a filename/folder
    location that you'll be able to find later.
}
write %default_config.txt config
unview
editor %default_config.txt
alert {Now choose a config file to use (most likely the file
    you just edited).}
do to-file request-file/title/file {
    Choose config file to use:} "File" %default_config.txt
view center-face layout gui
```

I added that code to the action block of the header image. Now the host can click the header to edit and save all the data for category, answer, question, and GUI size required to store complete Jeopardy sessions. I packaged this final program using XpackerX and gave it to my fiance. It suits her needs perfectly:

```
REBOL [title: "Jeopardy"]

config: {

    REBOL []
```

```
;_____________________________________________________________________

sizer: 4

Category-1: "Category 1"
Category-2: "Category 2"
Category-3: "Category 3"
Category-4: "Category 4"
Category-5: "Category 5"

answers: [
    "$100 Answer, Category 1"
    "$100 Answer, Category 2"
    "$100 Answer, Category 3"
    "$100 Answer, Category 4"
    "$100 Answer, Category 5"
    "$200 Answer, Category 1"
    "$200 Answer, Category 2"
    "$200 Answer, Category 3"
    "$200 Answer, Category 4"
    "$200 Answer, Category 5"
    "$300 Answer, Category 1"
    "$300 Answer, Category 2"
    "$300 Answer, Category 3"
    "$300 Answer, Category 4"
    "$300 Answer, Category 5"
    "$400 Answer, Category 1"
    "$400 Answer, Category 2"
    "$400 Answer, Category 3"
    "$400 Answer, Category 4"
    "$400 Answer, Category 5"
    "$500 Answer, Category 1"
    "$500 Answer, Category 2"
    "$500 Answer, Category 3"
    "$500 Answer, Category 4"
    "$500 Answer, Category 5"
]
questions: [
    "$100 Question, Category 1"
    "$100 Question, Category 2"
    "$100 Question, Category 3"
    "$100 Question, Category 4"
    "$100 Question, Category 5"
    "$200 Question, Category 1"
    "$200 Question, Category 2"
    "$200 Question, Category 3"
    "$200 Question, Category 4"
    "$200 Question, Category 5"
    "$300 Question, Category 1"
    "$300 Question, Category 2"
    "$300 Question, Category 3"
    "$300 Question, Category 4"
    "$300 Question, Category 5"
    "$400 Question, Category 1"
    "$400 Question, Category 2"
    "$400 Question, Category 3"
    "$400 Question, Category 4"
    "$400 Question, Category 5"
    "$500 Question, Category 1"
    "$500 Question, Category 2"
    "$500 Question, Category 3"
    "$500 Question, Category 4"
    "$500 Question, Category 5"
```

```
    ]

    ;_____________________________________________________________________

}

do config

header: load to-binary decompress 64#{
eJyVj3s804v/xz/bzGyFucRGohgm1ziuuU7CNGxWuTaxbBpyLVJUpItLcyfXNIZI
jkuJboqFJsNodG9uuXRyya18ncfj93t8//4+3/+9X6/H6/1+bY1ufQKQzg5ODgAI
BAIo2wNsjQE4QFQEKgoVERWFisJgomIIJAIBhyNkJSR3IuVlUSh5WTk5BSWMsoKi
6m45ORVtFVV1DawWFq2so6+jqY/RxGr+GwKCwWAIMYQMAiGjqSCnoPk/s/UMkBID
dAEmBLQPAEuBIFKgrZeAIgBAQP8C/D/bT4qIbS+3xUNIAASGgiFQsNhOKPRfEQSG
ANtlpKT3wmQMxGT32br7X5JHHTAKiGCW75JTNbQjkk6GZzYOqGl63Hl++UrSA952
jAIIAEH/e2AbMACCbMdsa5pSABgMgkBEwJD/c4DAAERKRNpAZq8tdJ+7PzHc7lL5
gy0BsHPbAJaCSAHWwNFKVxP98V4tOkWdfD7FEjacdjSkarU//C/n4IDH8tIsGPlr
F8jSOHVpqLEgNs++1taBHGuekzY4eM/ojEII4R/1Q+ZIEH2QWTpkVWn+ntRBthgZ
kympJr8jt/eRTxvTS1RNGD3meRMcm4kAZ2CAnTULVlJflfeI8BVvJWeJaNZitC7j
HSzwe21RvYdfbQEZcgNoufW05RxBPLziaGaLDnMiIu5cgjXhKavuH/3ewXQVtg6Q
BLgRCGfHVnFmn6LPF/fJDI1/TejRG5nB2X8vi0llhhEm3Fb/XnK+KeG/sKxkl6Py
E3ZteGrG4qqpYSazc72YsFrY6n0ht0oo2EEs4dhdJjJvOThxbRVx3ruRD921c4ct
XXp89nPCL1Ikh1ZwlkpvLRz53sKF8WfDpRW0V7dePB2671umDPyzp0PJxckwXbHc
pTnSOjTC54hLNXHdWIVdxdi9pDKhqNWOrCdNu3LPTh8e6tlxg9pQZw/KFHWY2OGM
Z33g/Y140axOK5pkwP2FSiCj0RhPPofjhXBtt52xdJjhjMZC9IMH+GNXzhlmhij6
fGjVfqN1lIHPvdV686aDXaIb6tn6gH1kYn1bpyM5Fi6c6cIsFn39lXJXLybMmRTv
soo4Nv7eqf/Byy2ANn2FByPXt1Xt8KZQEjhR08GMzINfIqIfBVT7HI4YKYlv/MRP
bv94k1JMSbSZATZOlnwEa64bdZBL/dZ6iEmlv1FwwpnNoCoplqla0drmB70Sd/H1
sZ9ijEDcfSc3TrdMbtAs3ZOg0+s96CofOOEnUod2Xqo2TylA/5xW1LtJzLNtp8vS
1UwD/YqprdJffrIwcdFhwvmr+CKmQDfT+P2R2AvojAzZ4RItnkwq00z74hIujcx1
Mzh42OioCXKvxeU3Ze4LhSJNyZYt72K7XrxlYf0p1TZX2MXlrulNBYgLUc2umSfJ
9fPvBnVHGUEaCRImNyxX74XvEyg79i/XIB/bfxsP1DCbf7O03rV/mvFg3u344qJ6
8IDbp3PMP47rJyasP1ii1wgX3yT4Zzh8dPztIlE1r3yH+6TqFHKa0ULlS3gTzBPE
oj4lDr4sEL9lrb7w2lG5jL3+g8+L+V5QmpvPdnGOED5HQNzNxvqubX52yVriGraq
tO6Jrj6r7imhjdzFQqA3yqLzMDmjNeevW4r1uGeg03qjg5oeh8z8EE0fVZ7+TelY
e3W27vblk7f1nIP9tWUx+PCfl50zdzXALUJVIesptLVwlles2qRi0C0KawvonDkS
0zL7bfgZvd+C5RxyaI+NmlSmpC36BtZYzzsVXrEgGDPrIzSG/uVbdKpq2DcrlPyi
N42GS+7PFLRyYffvkrYAZeof6SN31Ir1RJfPpgUQok53lbrErr+803Wn8lLXOyzc
bNkDgxT0n0PxnGhFDEeBBEs3nirrFOlQfMk1ZaKTK+lOZkzuhnKYwwE1xeMToRUa
m1188t34vrAtIIR9quLENPsjKebLtLCelN50PXG+jtoA74lKjV7Wr3Oc3Az4bqHH
fC+6atqPZVOY5d5DeF5My+8XnELv+nu97Rt08WoK2HySEbXy6rSNWuimAyaJS7A6
SX0ou0A3F0RV4b2KuCabPbAT8W5z54Vty8Izbu7qnrLNNLxwfLHB5liKIqWUNJ90
UfHge9e0PKqdZ/zrQxifKoPGRGGVylrq3QXxlbRaHMtvl1ezceO38aWZYhJPuMJH
ejSw/FUCKXPtlhuR899Au2LnD/fl/tDL75bbTwxWxWELbRTCUw+KEEO4+x4f+yK7
3NIWrBDsOyJweXaOzW5N2+4XZNVQ8GIlt3u37tUzZQKfQtWQdgLrxKIsAlpsf6tD
o/ZAtHwp++FX7iNdP1t88Jk0P6Rhaytfjllr5fos47hqjFXgyu3S1PEfY9U/P058
QWn5uLJ7wslybYFUsvopb0qbc/Nnm7CRPWdKT7sQuDHZpfn5uRG+T86/YcEu3Hrt
qNEbH/QR/R0a0GSdJJ3lFmDdk2+Vrw2R/IrOT3wZ3zliXC5Qzz5s5/XnCsqU3Ryv
RyQmaybczOGgx6kGS0+DCivDV/LMjJpNtDiBbQ1wOFQtcMMkZZ21r+++T2P02ZpG
riura80pOBzlxXNgZhgO0q4p3T6ePqo/9ojap3RR5kcx41Jp9ustIK5ctUlT/Zsb
SqOttfW3g+LDtANDVTJz3ZW78YyWXH4z+5MatFYdaTpNZtaOeCzr0gsqN9pkqlnB
aJE5SAUx9MS4hQRy8gt+7QTuz+hSoPD+StBsxmHcpWWzji2A+PlddStmXqvTld7D
It5HXH6KH6UkVzP4LFPesu165Bzn7PU8M+Eeca0dcPOOR8hk2tBMbjM1GnG6Th4P
bTaewuxw9UHqx6WXdQZ1QgV1Wfvt0esmCRs35On1p7W1RHe+rRTwm3eeS/YJwTX1
7seY44NCcNbGVr4UV7kQr2W7n+w+J+LEqePRR6qbW9KcsOJ3lRuuNGmaedb2fnia
6r+SRnLzuwMnTD0nWOqHS+iGR0zPl9NCc3fVwrTvo/6EMaQHown3/SQZuBn+85z0
A5Qn0pwjxFWWaw5bnzpBTHGiBzh3eZuO3bPe4ffgx8rpjoETl/GxCwpXrQ0rUNzI
/GzFqw/+umAS0bYoPoTMSieKOYANUqVjVOt/dJ12R8VWFsLYeUcT6hG/LbnCQaWu
nYh71cfoGmhOulEmjHajoedAYNc79Jq4fcZ13veLs3U0nW8efhb3IANWlUaGEXz6
SOqrnr5OXQ9lGobK/1kkW0UBoaYMwLUk0i5sSk1kr21+Yb53rXgFBIOdMpShR70h
l/ejIBs3xoZzsiML7GcHNCtryNM5U3u1WxKJDfcCM+MSQKBzcHP5l27zTafCJWtB
ODP3kcH3oNnHd6pC/nY/mV99Rv1tdJ6ClirctOvPIGhOXqMuuK+OWNgib6At5uzn
hDUjZBUh45SPH50uEPDDVJsQc3AvvfEGL+magCbq2xiep1ZX3aYTbenXi9+g4dfh
```

```
73l5xFvLRDrUvMFD1vLgr+e/bK6GIV6m/XJ7pBB8iLQo/iu63OmYtppIog038vOe
M0qCnEPH7Ds4AirpYAGrL/tc0y4lDdT3jHiSLyb1w0K38rcmQRJwaRHrX9ZPT6Y/
kzCWO9z6mFnaBuOHBH/rVhqrMcQXkE0JBz7nD3ziZVdpWE35yFO9Tq4Fz8T5lWjk
U1qQiR0I1OBDbpzMpLjyzuVxCgmbtOHcvflM3HK5lPbPNYf1oj2/U72/zl+UOsEj
SsbQVq/tcZMeKOMHTdJIxKZ3KaxBN7veuhaPRHBWpofxGGz2ksnNHPPlfguPGhfj
r/NsDNaRd5VnDHf8s5byWOa6TG5atR7hkLEfVbytUc6X6PR6eHpj4CyLau5BBrL2
d1wP2QJsHD7ouDj3hiskzdk2zN5mlIep3NXEgk5zuwNas4+s5hwvI9ck90b4FtBD
iSSWDqbMQ2IS6FIsEbJ9RxI+Y4V7c3HQSmqh4l4Vmbb0GTMNnrRjD8cyUiouJjHT
kyuWUsMHnnAXTzE3nkbF6X/2ezD1aHCjwNdz691/ABEeZGXYCwAA
}

do-button: func [num] [
    alert pick answers num
    alert pick questions num
    if find [1 2 3 4 5] num [val: $100]
    if find [6 7 8 9 10] num [val: $200]
    if find [11 12 13 14 15] num [val: $300]
    if find [16 17 18 19 20] num [val: $400]
    if find [21 22 23 24 25] num [val: $500]
    correct: request-list "Select:" ["Player 1 answered correctly"
        "Player 1 answered incorrectly" "Player 2 answered correctly"
        "Player 2 answered incorrectly" "Player 3 answered correctly"
        "Player 3 answered incorrectly" "Player 4 answered correctly"
        "Player 4 answered incorrectly"
    ]
    switch correct  [
        "Player 1 answered correctly" [
            player1/text: to-string ((to-money player1/text) + val)
            show player1
        ]
        "Player 1 answered incorrectly" [
            player1/text: to-string ((to-money player1/text) - val)
            show player1
        ]
        "Player 2 answered correctly" [
            player2/text: to-string ((to-money player2/text) + val)
            show player2
        ]
        "Player 2 answered incorrectly"[
            player2/text: to-string ((to-money player2/text) - val)
            show player2
        ]
        "Player 3 answered correctly" [
            player3/text: to-string ((to-money player3/text) + val)
            show player3
        ]
        "Player 3 answered incorrectly" [
            player3/text: to-string ((to-money player3/text) - val)
            show player3
        ]
        "Player 4 answered incorrectly"[
            player4/text: to-string ((to-money player4/text) - val)
            show player4
        ]
        "Player 4 answered correctly" [
            player4/text: to-string ((to-money player4/text) + val)
            show player4
        ]
    ]
]

view center-face layout gui: [
    tabs (sizer * 20)
```

```
backdrop effect [gradient 1x1 tan brown]
style button button effect [gradient blue blue/2] (
    to-pair rejoin [(20 * sizer) "x" (13 * sizer)]
) font [size: (sizer * 6)]
style box box brown (to-pair rejoin [(20 * sizer) "x" (7 * sizer
    )]) font [size: (sizer * 3)]
image header (to-pair rejoin [(132 * sizer) "x" (8 * sizer)]) [
    contin: request/confirm {
        This will end the current game.  Continue?}
    if contin = false [break]
    loadoredit:  request/confirm "Load previously edited config file?"
    if loadoredit = true [
        do to-file request-file/title/file {
            Choose config file to use:} "File" %default_config.txt
        unview
        view center-face layout gui
        break
    ]
    alert {Edit carefully, maintaining all quotation marks.
        You can open a previously saved file if needed.
        When done, click SAVE-AS and then QUIT.
        Be sure choose a filename/folder
        location that you'll be able to find later.
    }
    write %default_config.txt config
    unview
    editor %default_config.txt
    alert {Now choose a config file to use (most likely the file
        you just edited).}
    do to-file request-file/title/file {
        Choose config file to use:} "File" %default_config.txt
    view center-face layout gui
]
space (to-pair rejoin [(8 * sizer) "x" (2 * sizer)])
pad (sizer * 2)
across
box (to-pair rejoin [(132 * sizer) "x" (2 * sizer)]
    ) effect [gradient 1x0 brown black]
return
box Category-1
box Category-2
box Category-3
box Category-4
box Category-5
return
box (to-pair rejoin [(132 * sizer) "x" (2 * sizer)]
    ) effect [gradient 1x0 brown black]
return
button "$100" [face/feel: none face/text: "" do-button 1]
button "$100" [face/feel: none face/text: "" do-button 2]
button "$100" [face/feel: none face/text: "" do-button 3]
button "$100" [face/feel: none face/text: "" do-button 4]
button "$100" [face/feel: none face/text: "" do-button 5]
return
button "$200" [face/feel: none face/text: "" do-button 6]
button "$200" [face/feel: none face/text: "" do-button 7]
button "$200" [face/feel: none face/text: "" do-button 8]
button "$200" [face/feel: none face/text: "" do-button 9]
button "$200" [face/feel: none face/text: "" do-button 10]
return
button "$300" [face/feel: none face/text: "" do-button 11]
button "$300" [face/feel: none face/text: "" do-button 12]
button "$300" [face/feel: none face/text: "" do-button 13]
```

```
    button "$300" [face/feel: none face/text: "" do-button 14]
    button "$300" [face/feel: none face/text: "" do-button 15]
    return
    button "$400" [face/feel: none face/text: "" do-button 16]
    button "$400" [face/feel: none face/text: "" do-button 17]
    button "$400" [face/feel: none face/text: "" do-button 18]
    button "$400" [face/feel: none face/text: "" do-button 19]
    button "$400" [face/feel: none face/text: "" do-button 20]
    return
    button "$500" [face/feel: none face/text: "" do-button 21]
    button "$500" [face/feel: none face/text: "" do-button 22]
    button "$500" [face/feel: none face/text: "" do-button 23]
    button "$500" [face/feel: none face/text: "" do-button 24]
    button "$500" [face/feel: none face/text: "" do-button 25]
    return
    box (to-pair rejoin [(132 * sizer) "x" (2 * sizer)]
        ) effect [gradient 1x0 brown black]
    return tab
    box "Player 1:" effect [gradient 1x1 tan brown]
    player1: box white "$0" font [color: black size: (sizer * 4)] [
        face/text: request-text/title/default "Enter Score:" face/text
    ]
    box "Player 2:" effect [gradient 1x1 tan brown]
    player2: box white "$0" font [color: black size: (sizer * 4)] [
        face/text: request-text/title/default "Enter Score:" face/text
    ]
    return tab
    box "Player 3:" effect [gradient 1x1 tan brown]
    player3: box white "$0" font [color: black size: (sizer * 4)] [
        face/text: request-text/title/default "Enter Score:" face/text
    ]
    box "Player 4:" effect [gradient 1x1 tan brown]
    player4: box white "$0" font [color: black size: (sizer * 4)] [
        face/text: request-text/title/default "Enter Score:" face/text
    ]
]
```

10.9 Case 9 - Creating a Tetris Game Clone

One of my favorite games to play is Tetris. I particularly like the "Rebtris" clone, written in REBOL by Frank Sievertsen. While playing Rebtris recently, it struck me that writing a Tetris clone of my own would be a helpful exercise for this tutorial. In conceiving how to make it a bit different from all the other endless variations of Tetris, I thought "why not try a text version?". Creating it may be easier than a GUI version, and it would run on machines where GUI versions of REBOL aren't available. Writing a text version of Tetris would also force me to organize a set of display techniques that could be useful in laying out other text-based applications. Sounds like a fun project with some useful side effects.

NOTE: I've never considered how to build a Tetris clone, and I'm writing this section as I design, experiment, and write code for the program. So as you read this, you'll follow my exact train of thought in putting the program together, and I'll make no attempt to artificially clean up the process. The point of this tutorial is to demonstrate exactly how to go about creating all types of programs on your own. I hope that following my footsteps exactly - imperfections and all - should be a helpful experience. Let's see what happens...

Instead of starting this entire thing from scratch, I remembered briefly skimming a tutorial about how to create a text positioning dialect called "TUI". I found it again at http://www.rebolforces.com/articles/tui-dialect/ and looked for some reusable code...

Here's the TUI dialect, created by Ingo Hohmann (I renamed his "cursor2" function to "tui"):

```
tui: func [
    {Cursor positioning dialect (iho)}
    [catch]
    commands [block!]
    /local screen-size string arg cnt cmd c err
][
    screen-size: (
        c: open/binary/no-wait [scheme: 'console]
        prin "^(1B)[7n"
        arg: next next to-string copy c
        close c
        arg: parse/all arg ";R"
        forall arg [change arg to-integer first arg]
        arg: to-pair head arg
    )
    string: copy ""
    cmd: func [s][join "^(1B)[" s]
    if error? set/any 'err try [
        commands: compose bind commands 'screen-size ][
        throw err
    ]
    arg: parse commands [
        any [
            'direct set arg string! (append string arg) |
            'home  (append string cmd "H") |
            'kill  (append string cmd "K") |
            'clear (append string cmd "J") |
            'up    set arg integer! (append string cmd [
                arg "A"]) |
            'down  set arg integer! (append string cmd [
                arg "B"]) |
            'right set arg integer! (append string cmd [
                arg "C"]) |
            'left  set arg integer! (append string cmd [
                arg "D"]) |
            'at    set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
```

```
            'del   set arg integer! (append string cmd [
                arg "P"]) |
            'space set arg integer! (append string cmd [
                arg "@"]) |
            'move  set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
            set cnt integer! set arg string! (
                append string head insert/dup copy "" arg cnt
            ) |
            set arg string! (append string arg)
        ]
        end
    ]
    if not arg [throw make error! "Unable to parse block"]
    string
]
```

I read the tutorial and played with the code a bit. In addition to being a great tutorial, the end product is a useful tool for formatting output in console applications. The function "tui" gets passed a block of parameters including text to be printed, directional keywords to move the cursor around the screen ("home", "up", "down", "left", "right", and "at" a specific location) and several other commands to get the screen size, clear the screen, delete text, repeat text, and insert spaces.

I tried a few commands to get familiar with the syntax:

```
prin tui [ clear ]
print tui [ 50  "-" ]
print tui [ right 10 down 7 50  "x" ]
prin tui [ clear right 10 down 10 50  "x" ]
print tui [ clear home "message1"]
print tui [ home space 20 "message2"]
print tui [ at 20x20 "message3" kill "message4"]
print tui [ at 20x20  del 10]
print tui [ move 10x10]
prin tui [ clear (screen-size/y * screen-size/x - 4)  "x" ]
```

I had more fun playing with the TUI dialect than I did playing Tetris :) Basically, TUI wraps up some of the native print control codes built into REBOL, in a nice clean format that eliminates all the odd characters used in native codes. It contains everything required to move game pieces around the screen, so I'll start to come up with some requirements to build the game. Here's an outline that covers the main design goals I'm conceiving at this point:

1. Draw a static playing field (the unchanging graphic backdrop design that's on screen the whole time the game is being played). This will represent the left, right, and lower vertical bounds which the game coordinates may not exceed.
2. There are 7 block shapes used in the game. Create text versions of each graphic shape, in each of the 4 possible rotated positions. Come up with code to print and delete each of the graphic shapes. The TUI dialect will let me print and delete characters anywhere on the screen. I'll use directional statements to print the required characters, starting from any given coordinate. Put all these shape routines into a block for easy naming and reuse.
3. Write a continuous loop to put one shape on the screen, make it fall at a given speed, and allow the user to spin it around and move it left-right.
4. If a shape touches the bottom of the playing field, make it lock into the grid of other shapes that have already fallen. If the bottom row is complete, remove it, and make all the rows above it fall down a row to take its place. If the shape touches the ceiling, end the game.

The first part of the outline is easy. I'll use the "print a-line" code created in the "loops and conditions - a simple data storage app" example earlier in this tutorial. Here's a simple little backdrop that's printed to the screen:

```
a-line: copy [] loop 28 [append a-line " "]
a-line: rejoin ["   |"  to-string a-line "|"]
loop 30 [print a-line] prin "   " loop 30 [prin "+"] print ""
```

For the second part, I need to create some code to print the 7 block shapes. They look like this:

```
;   1   ####    2   ###     3   ###    4    ###
;                    #          #             #
;
;   5   ##      6    ##     7   ##
;        ##         ##          ##
```

Here are all the possible variations when the above shapes are rotated clockwise:

```
;                   #
;   1   ####    2   #
;                   #
;                   #
;
;   3   ###     4    #      5    #      6   #
;        #           ##         ###         ##
;                    #                      #
;
;   7   ###     8   ##      9     #     10  #
;       #            #          ###         #
;                    #                      ##
;
;   11  ###     12   #      13  #       14  ##
;         #          #          ###         #
;                   ##                      #
;
;   15  ##      16   #
;        ##         ##
;                   #
;
;   17   ##     18  #
;       ##          ##
;                    #
;
;   19  ##
;       ##
```

To print any piece, I can start at the top left coordinate in the shape and move the appropriate number of spaces right, left, and/or down to print the other characters in each piece. Starting at the first character in shape 2, for example, I would move as follows: "#", down 1 left 1, "#", down 1 left 1, "#", down 1 left 1, "#". Shape 3 would move as follows: "###", down 1 left 2, "#". Here's a block called "shape", made up of individual blocks that can be passed to tui to print all the above shapes:

```
shape: [
    ["####"]
    ["#" down 1 left 1 "#" down 1 left 1 "#" down 1 left 1 "#"]
    ["###" down 1 left 2 "#"]
    [right 1 "#" down 1 left 2 "##" down 1 left 1 "#"]
    [right 1 "#" down 1 left 2 "###"]
    ["#" down 1 left 1 "##" down 1 left 2 "#"]
    ["###" down 1 left 3 "#"]
    ["##" down 1 left 1 "#" down 1 left 1 "#"]
```

```
    [right 2 "#" down 1 left 3 "###"]
    ["#" down 1 left 1 "#" down 1 left 1 "##"]
    ["###" down 1 left 1 "#"]
    [right 1 "#" down 1 left 1 "#" down 1 left 2 "##"]
    ["#" down 1 left 1 "###"]
    ["##" down 1 left 2 "#" down 1 left 1 "#"]
    ["##" down 1 left 1 "##"]
    [right 1 "#" down 1 left 2 "##" down 1 left 2 "#"]
    [right 1 "##" down 1 left 3 "##"]
    ["#" down 1 left 1 "##" down 1 left 1 "#"]
    ["##" down 1 left 2 "##"]
]
```

Now I can use the format "prin tui shape/number" to print any shape. For example:

```
prin tui shape/3
```

is the same as writing:

```
prin tui [right 1 "#" down 1 left 2 "##" down 1 left 1 "#"]
```

I came up with the following code to print out each shape, to check for errors, and to get used to using the above format. Notice the use of the "compose" function:

```
for i 1 19 1 [
    print tui [clear]
    print rejoin ["shape " i ":"]
    do compose [print tui shape/(i)]
    ask ""
]
```

To erase the shapes, I decided to extend the block using duplicates of each shape to print spaces instead of "#"s. Now all I have to do is add 19 to any shape's index number, and I can print out a shape made of spaces that erases the original shape made of "#"s. Here's the final shape block:

```
shape: [
    ["####"]
    ["#" down 1 left 1 "#" down 1 left 1 "#" down 1 left 1 "#"]
    ["###" down 1 left 2 "#"]
    [right 1 "#" down 1 left 2 "##" down 1 left 1 "#"]
    [right 1 "#" down 1 left 2 "###"]
    ["#" down 1 left 1 "##" down 1 left 2 "#"]
    ["###" down 1 left 3 "#"]
    ["##" down 1 left 1 "#" down 1 left 1 "#"]
    [right 2 "#" down 1 left 3 "###"]
    ["#" down 1 left 1 "#" down 1 left 1 "##"]
    ["###" down 1 left 1 "#"]
    [right 1 "#" down 1 left 1 "#" down 1 left 2 "##"]
    ["#" down 1 left 1 "###"]
    ["##" down 1 left 2 "#" down 1 left 1 "#"]
    ["##" down 1 left 1 "##"]
    [right 1 "#" down 1 left 2 "##" down 1 left 2 "#"]
    [right 1 "##" down 1 left 3 "##"]
    ["#" down 1 left 1 "##" down 1 left 1 "#"]
    ["##" down 1 left 2 "##"]
```

```
    ; Here are the same shapes, with spaces instead of "#"s:

    ["    "]
    [" " down 1 left 1 " " down 1 left 1 " " down 1 left 1 " "]
    ["   " down 1 left 2 " "]
    [right 1 " " down 1 left 2 "  " down 1 left 1 " "]
    [right 1 " " down 1 left 2 "   "]
    [" " down 1 left 1 "  " down 1 left 2 " "]
    ["   " down 1 left 3 " "]
    ["  " down 1 left 1 " " down 1 left 1 " "]
    [right 2 " " down 1 left 3 "   "]
    [" " down 1 left 1 " " down 1 left 1 "  "]
    ["   " down 1 left 1 " "]
    [right 1 " " down 1 left 1 " " down 1 left 2 "  "]
    [" " down 1 left 1 "   "]
    ["  " down 1 left 2 " " down 1 left 1 " "]
    ["  " down 1 left 1 "  "]
    [right 1 " " down 1 left 2 "  " down 1 left 2 " "]
    [right 1 "  " down 1 left 3 "  "]
    [" " down 1 left 1 "  " down 1 left 1 " "]
    ["  " down 1 left 2 "  "]
]
```

I wrote another quick script to test it. Notice the "i + 19" used to erase the exiting shape:

```
for i 1 19 1 [
    print tui [clear]
    print rejoin ["shape " i ":"]
    do compose [prin tui [move 10x10] print tui shape/(i)]
    ask ""
    do compose [prin tui [move 10x10] print tui shape/(i + 19)]
    print rejoin ["shape " i " has been erased."]
    ask ""
]
```

Beautiful. Steps 1 and 2 are complete. Now I can work on the last part of the outline (the things related to how the game actually plays, moving pieces and responding to user input). First, I'll get the shapes to fall down the screen. That'll be done by printing, erasing and then redrawing the piece one row lower, in a continuous loop. Here's an outline to organize that thought process:

1. Start by clearing the screen.
2. Pieces appear in a random order, so come up with a random number to represent some random shape's index number.
3. Use a for loop to increment the vertical position of the piece: for each row, print the random piece number at the current horizontal position (initially set to 15), and at the vertical position represented by the current "for" variable.
4. Wait a moment, then erase the piece (using the shape number + 19). Then increment the row number and start again.
5. When the piece reaches the last row, print it there without erasing.
6. Wrap that whole thing in a forever loop to keep it going indefinitely.

Let's get that much going:

```
prin tui [clear]

forever [
    random/seed now
    r: random 19      ; the number of a random shape
    xpos: 18          ; the initial horizontal position
```

```
    for i 1 25 1 [
        pos: to-pair rejoin [i "x" xpos]
        ; print the shape represented by "r" at the "pos"
        ; coordinate:
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        ; The wait time could be a user controlled variable, or
        ; it could be sped up as the difficulty level increases:
        wait :00:00.30
        ; erase the shape, then continue the loop:
        do compose/deep [
            prin tui [at (pos)] print tui shape/(r + 19)]
    ]
    ; reprint the shape at its final resting place:
    do compose/deep [prin tui [move (pos)] print tui shape/(r)]
]
```

NOTE: It struck me in writing the above code that the TUI function actually takes all its coordinates in an unusual order. Typically, in coordinates with the form XxY, "X" is the horizontal position and "Y" is the vertical position. TUI uses the format YxX, where Y is the vertical position measured in rows from the top of the screen. X is the horizontal position, measured in columns from the left side of the screen. Keep in mind that the order of X and Y coordinates is opposite the normal expectation.

Now I need to come up with a way for the user to control the horizontal position of the shape. Here's some pseudo code to help me think about how to do that:

1. Be on the lookout for keystroke input from the user.
2. If the user presses the "l" key, add 1 to the current horizontal position of the shape (held in the variable "xpos"). If the user presses the "k" key, subtract 1 from xpos.

First, I need a way to get keystroke input without blocking the program flow (i.e., I need to wait for keystroke input to be acknowledged when it occurs, but I can't just stop the normal program flow to wait for key presses. For game play to continue, the "for" and "forever" loops can't be interrupted. So I searched Google for "REBOL key stroke" and got pointed to the following code at http://www.rebol.org/cgi-bin/cgiwrap/rebol/ml-display-thread.r?m=rmlSCRQ (in the REBOL mailing list archive):

```
c: open/binary/no-wait [scheme: 'console]
; set following to whatever you wish
; intentionally slow at 2 secs so you can "see" the effect
wait-duration: :0:2
d: 0
forever [
    if not none? wait/all [c wait-duration] [
        print to-char to-integer copy c
    ]
    d: d + 1 ;let's do other stuff
    print d
]
```

That little bit of code does exactly what I need. The parts required for my needs are:

```
c: open/binary/no-wait [scheme: 'console]
forever [
    if not none? wait/all [c wait-duration] [
        print to-char to-integer copy c
    ]
]
```

I adjusted the variable names, checked for "k" or "l" key presses, and used the code below to test

that it worked the way I wanted:

```
keys: open/binary/no-wait [scheme: 'console]
forever [
    if not none? wait/all [keys :00:00.01] [
        switch to-string to-char to-integer copy keys [
            "k" [print "you pressed k"]
            "l" [print "you pressed l"]
        ]
    ]
; print "nothing pressed" ; make sure it's working
]
```

Next, I integrated the above code into the loop created earlier to drop the shape down the screen. Notice that I added a conditional "if", to be executed when either "k" or "l" keystrokes are encountered. It checks that the horizontal bounds don't go outside the 5-30 positions. That keeps the shapes within the horizontal boundaries of the playing field. Also, notice that the variable "old-xpos" is used to hold the position of the shape that needs to be erased:

```
keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 18
    for i 1 25 1 [
        pos: to-pair rejoin [i "x" xpos]
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        old-xpos: xpos
        if not none? wait/all [keys :00:00.30] [
            switch to-string to-char to-integer copy keys [
                "k" [if (xpos > 5) [xpos: xpos - 1]]
                "l" [if (xpos < 30) [xpos: xpos + 1]]
            ]
        ]
        pos: to-pair rejoin [i "x" old-xpos]
        do compose/deep [
            prin tui [at (pos)] print tui shape/(r + 19)]
    ]
    do compose/deep [prin tui [move (pos)] print tui shape/(r)]
]
```

It's coming along well :) Now I need to be able to spin the shapes around. Here's some pseudo code to organize my thoughts:

1. Watch for the "O" key to be pressed. That will be the keycode to run the shape spinning code.
2. Create a set of conditionals to cycle through the list of rotated shapes related to the current shape. For example, if the current shape (variable "r") is number 12, then the rotated versions of that shape are numbers 11-14. With each press of the "O" key, replace the variable r with the next shape in that list. That logic must "wrap around" (i.e., the next shape after 14 should be 11). Instead of using a block list of shapes to do this, I decide to use a switch structure to individually map each shape to the one it should rotate to (something like "if shape r is now #14, turn shape r into #11" - do that explicitly for each shape).

I already have some code to watch for keystrokes, so I'll try the last part of the above outline first:

```
switch to-string r [
    "1" [r: 2]
    "2" [r: 1]
```

```
    "3" [r: 4]
    "4" [r: 5]
    "5" [r: 6]
    "6" [r: 3]
    "7" [r: 8]
    "8" [r: 9]
    "9" [r: 10]
    "10" [r: 7]
    "11" [r: 12]
    "12" [r: 13]
    "13" [r: 14]
    "14" [r: 11]
    "15" [r: 16]
    "16" [r: 15]
    "17" [r: 18]
    "18" [r: 17]
    "19" [r: 19]
]
```

Wait a sec - that makes the shapes rotate clockwise (from #11 go to #12, #14 to #11, etc.) I prefer for them to rotate counterclockwise (#11 to #14, #14 to #13, etc). Here's the revised code:

```
switch to-string r [
    "1" [r: 2]
    "2" [r: 1]
    "3" [r: 6]
    "4" [r: 3]
    "5" [r: 4]
    "6" [r: 5]
    "7" [r: 10]
    "8" [r: 7]
    "9" [r: 8]
    "10" [r: 9]
    "11" [r: 14]
    "12" [r: 11]
    "13" [r: 12]
    "14" [r: 13]
    "15" [r: 16]
    "16" [r: 15]
    "17" [r: 18]
    "18" [r: 17]
    "19" [r: 19]
]
```

Now add the letter "O" to the list of keys to be watched, and run the above code when it's pressed. Also create an "old-r" variable to retain the number of the shape that needs to be erased. (Since the user changes shapes after the current one has been printed, we need to keep track of which one to erase):

```
keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 18
    for i 1 25 1 [
        pos: to-pair rejoin [i "x" xpos]
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        old-xpos: xpos
        old-r: r
```

```
        if not none? wait/all [keys :00:00.30] [
            keystroke: to-string to-char to-integer copy keys
            switch keystroke [
                "k" [if (xpos > 5) [xpos: xpos - 1]]
                "l" [if (xpos < 30) [xpos: xpos + 1]]
                "o" [switch to-string r [
                    "1" [r: 2]
                    "2" [r: 1]
                    "3" [r: 6]
                    "4" [r: 3]
                    "5" [r: 4]
                    "6" [r: 5]
                    "7" [r: 10]
                    "8" [r: 7]
                    "9" [r: 8]
                    "10" [r: 9]
                    "11" [r: 14]
                    "12" [r: 11]
                    "13" [r: 12]
                    "14" [r: 13]
                    "15" [r: 16]
                    "16" [r: 15]
                    "17" [r: 18]
                    "18" [r: 17]
                    "19" [r: 19]
                ]]
            ]
        ]
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r + 19)
        ]
    ]
    do compose/deep [prin tui [at (pos)] print tui shape/(r)]
]
```

The shapes are moving correctly now, but there's still a lot of work to be done. The first line of the last section of the overall game outline reads: "If the shape touches the bottom of the playing field, make it lock into the grid of other shapes that have already fallen". Right now the pieces all just fall to different stopping points in the playing field (depending on their height), and they don't stack on top of each other. Here's some pseudo code to fix that:

1. I need to be aware of the highest coordinate in each column on the playing field. When the game starts, the highest coordinate in every column of the playing field is row 30 (the flat bottom line that makes up the playing field). I'll store each of these coordinates in a block called "floor".
2. I also need to be aware of the lowest coordinate in each column of the currently falling shape. I'll make a block called "edge" to hold those coordinates (referring to the lower edges of the shape). Those coordinates will define the position of each of the lowest points in the currently falling shape, in relation to its top left point (the "pos" coordinate).
3. Every time the shape falls one position down the screen, add each of the edge coordinates to the pos coordinate. If any of those coordinates is one position higher than the floor coordinate in the same column, then stop moving that shape (break out of the "for" loop that makes the shape fall). Use a foreach loop to cycle through the current coordinates in the relevant columns of each block, performing a comparison check on the floor and edge coordinates in each column.
4. When a shape finishes its drop down the screen, calculate the new highest position in the columns it occupies (the coordinates of the top character in each column), and make those changes to the block that holds the high point information. To do that, I'll need to make a "top" block to hold the relative positions of the highest coordinates in the shape, and add them to the height of the current coordinates in the appropriate columns.

I'll start out simply, just getting each shape to lay flat on the floor of the playing field (row 30). For the

moment, all I need to do is create a block of floor coordinates that represents that bottom line:

```
floor:     [30x1 30x2 30x3 30x4 30x5 30x6 30x7 30x8 30x9 30x10 30x11
    30x12 30x13 30x14 30x15 30x16 30x17 30x18 30x19 30x20 30x21
    30x22 30x23 30x24 30x25 30x26 30x27 30x28 30x29 30x30 30x31
    30x32 30x33 30x34 30x35]
```

Next, I'll define a set of lower coordinates for each shape, and store them in a nested block structure similar to the earlier "shape" block. "0x0" refers to the same coordinate as "pos" (0 positions to the right, and 0 positions down from "pos"). "0x10" is one position to the right, and "1x0" is one position down. I look at the visual representations of the shapes again to come up with the list:

```
;                  #
;   1   ####    2  #
;                  #
;                  #
;
;   3   ###     4   #      5    #      6   #
;        #          ##         ###         ##
;                    #                     #
;
;   7   ###     8  ##      9     #     10  #
;       #           #          ###         #
;                   #                      ##
;
;   11  ###     12  #      13  #       14  ##
;         #         #          ###         #
;                  ##                     #
;
;   15  ##      16  #
;        ##         ##
;                   #
;
;   17   ##     18  #
;       ##          ##
;                    #
;
;   19  ##
;       ##
```

Here's the complete set of low point definitions for each shape:

```
edge: [ [0x0 0x1 0x2 0x3] [3x0] [0x0 1x1 0x2] [1x0 2x1]
    [1x0 1x1 1x2] [2x0 1x1] [1x0 0x1 0x2] [0x0 2x1] [1x0 1x1 1x2]
    [2x0 2x1] [0x0 0x1 1x2] [2x0 2x1] [1x0 1x1 1x2] [2x0 0x1]
    [0x0 1x1 1x2] [2x0 1x1] [1x0 1x1 0x2] [1x0 2x1] [1x0 1x1] ]
```

So, the relative coordinates of the low points in shape 3, for example, are referred to as edge/3. Here's some sample code to demonstrate how I can now refer to the bottom points in any shape using a foreach loop. The code "pos + position" refers to the low edge in each column:

```
pos: 5x5
r: 6
foreach position compose edge/(r) [print pos + position]
```

To check if any of those edges are touching the floor, use a foreach loop to cycle through the current coordinates in the relevant columns of each block, performing a comparison check on the floor and

edge coordinates in each column. Here's some sample code to flesh out and test that idea:

```
pos: 30x10
for r 1 19 1 [
    print tui [clear]
    prin "Piece: " print r
    foreach po compose edge/(r) [
        print pos + po
        foreach coord floor [
            floor-y: to-integer first coord
            floor-x: to-integer second coord
            edge-y: to-integer first pos + to-integer first po
            edge-x: to-integer second pos + to-integer second po
            print rejoin [
                "edge: " edge-y "x" edge-x " "
                "floor: "floor-y "x" floor-x
            ]
            if (edge-y >= floor-y) and (floor-x = edge-x) [
                print rejoin [
                    "You're touching or beyond the floor at: "
                    pos + po
                ]
            ]
        ]
    ]
    ask ""
]
```

Now let's integrate this technique into the existing code. We'll use a new variable "stop" to break out of the loop that drops the shape, when the current shape touches the floor:

```
keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 18
    for i 1 32 1 [
        pos: to-pair rejoin [i "x" xpos]
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        old-r: r
        old-xpos: xpos
        if not none? wait/all [keys :00:00.30] [
            keystroke: to-string to-char to-integer copy keys
            switch keystroke [
                "k" [if (xpos > 5) [xpos: xpos - 1]]
                "l" [if (xpos < 30) [xpos: xpos + 1]]
                "o" [switch to-string r [
                    "1" [r: 2]
                    "2" [r: 1]
                    "3" [r: 6]
                    "4" [r: 3]
                    "5" [r: 4]
                    "6" [r: 5]
                    "7" [r: 10]
                    "8" [r: 7]
                    "9" [r: 8]
                    "10" [r: 9]
                    "11" [r: 14]
                    "12" [r: 11]
                    "13" [r: 12]
```

```
                    "14" [r: 13]
                    "15" [r: 16]
                    "16" [r: 15]
                    "17" [r: 18]
                    "18" [r: 17]
                    "19" [r: 19]
                ]]
            ]
        ]
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r + 19)
        ]
        stop: false
        foreach po compose edge/(r) [
            foreach coord floor [
                floor-y: to-integer first coord
                floor-x: to-integer second coord
                edge-y:  i + to-integer first po
                edge-x:  xpos + to-integer second po
                if (edge-y >= floor-y) and (floor-x = edge-x) [
                    stop: true
                    break
                ]
            ]
        ]
        if stop = true [break]
    ]
    do compose/deep [prin tui [at (pos)] print tui shape/(old-r)]
]
```

This works, but there's a bug. If the piece has been spun around (using the "O" key), the new foreach loop fails to stop the piece from falling. That's because the foreach loop only cycles through the coordinates of the "edge/r" block. If the user flips the shape around, the "r" value gets changed before this code is run. The easiest way to fix this problem is to simply repeat the foreach loop using the "edge/old-r" block. This is an inefficient quick hack, but I'm writing this late at night - and there's some value to pointing out bad coding practice - so I choose to use that solution. I make a promise to myself to come up with a more elegant solution later... (Note to self: once a coding solution has been implemented, changes are harder to make, and bad code typically remains permanent ... I need to be careful about using quick hacks). Here's the current code:

```
keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 18
    for i 1 32 1 [
        pos: to-pair rejoin [i "x" xpos]
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        old-r: r
        old-xpos: xpos
        if not none? wait/all [keys :00:00.30] [
            keystroke: to-string to-char to-integer copy keys
            switch keystroke [
                "k" [if (xpos > 5) [xpos: xpos - 1]]
                "l" [if (xpos < 30) [xpos: xpos + 1]]
                "o" [switch to-string r [
                    "1" [r: 2]
                    "2" [r: 1]
                    "3" [r: 6]
                    "4" [r: 3]
                    "5" [r: 4]
```

```
                    "6" [r: 5]
                    "7" [r: 10]
                    "8" [r: 7]
                    "9" [r: 8]
                    "10" [r: 9]
                    "11" [r: 14]
                    "12" [r: 11]
                    "13" [r: 12]
                    "14" [r: 13]
                    "15" [r: 16]
                    "16" [r: 15]
                    "17" [r: 18]
                    "18" [r: 17]
                    "19" [r: 19]
                ]]
            ]
        ]
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r + 19)
        ]
        stop: false
        foreach po compose edge/(r) [
            foreach coord floor [
                floor-y: to-integer first coord
                floor-x: to-integer second coord
                edge-y:  i + to-integer first po
                edge-x:  xpos + to-integer second po
                if (edge-y = floor-y) and (floor-x = edge-x) [
                    stop: true
                    break
                ]
            ]
        ]
        foreach po compose edge/(old-r) [
            foreach coord floor [
                floor-y: to-integer first coord
                floor-x: to-integer second coord
                edge-y:  i + to-integer first po
                edge-x:  old-xpos + to-integer second po
                if (edge-y = floor-y) and (floor-x = edge-x) [
                    stop: true
                    break
                ]
            ]
        ]
        if stop = true [break]
    ]
    do compose/deep [prin tui [at (pos)] print tui shape/(old-r)]
]
```

Next, I decide to test the existing program for other bugs. I've been keeping separate text files containing all the code changes I make as I go along. Every time I make, test, and change a chunk of code, I save the new trial version with a new filename and version number. I save each version, just so that I don't permanently erase old code with each change - it may be potentially useful. My current working version is now #19.

I noticed during this debugging session that shape 1 still breaks through the right side of the wall. I could change that by adjusting the "(xpos < 30)" conditional expression that occurs when the "L" key gets pressed. But that solution will keep the other shapes from laying snugly against the wall. In fact, that additional problem is occurring now with shapes that are only 2 characters wide - I didn't notice until now. To deal with these problems, I create a block of values called "width", listing the widths of all 19 shapes, which can be used in the existing conditional expression:

```
width: [4 1 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 2]
```

Now I can check if the shape is at the right boundary, using the revised code below:

```
[if (xpos < (33 - compose width/(r))) [
    xpos: xpos + 1]
]
```

That check also needs to be performed every time the "O" key is pressed (we don't want the shape breaking out of the wall when it spins). I make the above changes to my current version of the program, and the problems are fixed.

The game is really starting to take shape! Now we need to make the shapes stack on top of each other. Earlier, I wrote these outline thoughts: "when a shape finishes its drop down the screen, calculate the new highest position in the columns it occupies (the coordinates of the top character in each column), and make those changes to the block that holds the high point information. To do that, I'll need to make a "top" block to hold the relative positions of the highest coordinates in the shape, and add them to the height of the current coordinates in the appropriate columns". Sounds like I'll need to loop through some columns to make the changes to the floor.

To create the "top" block I look at the visual representations of each shape once again, and come up with a coordinate list representing the high points in the shape, relative to the top left coordinate. It's similar to the "edge" block:

```
top: [ [0x0 0x1 0x2 0x3] [0x0] [0x0 0x1 0x2] [1x0 0x1]
    [1x0 0x1 1x2] [0x0 1x1] [0x0 0x1 0x2] [0x0 0x1] [1x0 1x1 0x2]
    [0x0 2x1] [0x0 0x1 0x2] [2x0 0x1] [0x0 1x1 1x2] [0x0 0x1]
    [0x0 0x1 1x2] [1x0 0x1] [1x0 0x1 0x2] [0x0 1x1] [0x0 0x1] ]
```

The shape finishes its drop down the screen during the previous foreach loops we created, so to calculate the new highest positions in the columns occupied by the shape, I first need to determine which shape was the last one on the screen ("r" or "old-r"). The quick hack I made earlier is now coming back to bite me a bit - I now need to make duplicates of any changes that occur in both foreach loops:

```
stop-shape-num: r
; (or stop-shape-num: old-r, depending on the foreach loop)
stop: true
break
```

Now to make the changes to the "floor" block, I loop through the columns occupied by the piece, setting each of the top characters in the shape to be the high coordinates in the respective columns of the floor. The "poke" function lets me replace the original coordinates in the floor block with the new coordinates. Those changes are made just before breaking out of the loop that drops the shape:

```
if stop = true [
    ; get the left-most column the last shape occupies:
    left-col: second pos
    ; get the number of columns the shape occupies:
    width-of-shape: length? compose top/(stop-shape-num)
    ; get the right most column the shape occupies:
    right-col: left-col + width-of-shape - 1
    ; Loop through each column occupied by the shape,
    ; replacing each coordinate in the current column
    ; of the floor with the new high coordinate:
```

```
    counter: 1
    for current-column left-col right-col 1 [
        add-coord: compose top/(stop-shape-num)/(counter)
        new-floor-coord: (pos + add-coord + -1x0)
        poke floor current-column new-floor-coord
        counter: counter + 1
    ]
    break
]
```

The new stacking code works, but there's a design flaw. If I maneuver a shape into an unoccupied space directly underneath any high point in the floor, without first touching the high point in that column, the piece doesn't stop. Furthermore, if that happens, it changes the new high point to the bottom of the column which the current shape occupies. I realize here that what I need to mark are not only the high points in the floor, but also every additional coordinate on the screen that contains a character. This is just as easy to accomplish. Instead of *changing* the current coordinates in the floor block (using the "poke" function):

```
poke floor current-column new-floor-coord
```

just *add* the new coordinates to the list (using "append"). That will keep track of all points at which a character is printed on the screen:

```
append floor new-floor-coord
```

That fixes the problem above, but I've also realized that if I move a shape sideways into an open position in the floor, the characters sometimes still overlap inappropriately. That's because the "top" and "edge" blocks only mark the highest and lowest points in each shape. It strikes me now that I could just combine those two blocks into one, marking all the coordinates occupied by a shape. Here's the new block - I call it "oc", short for "occupied":

```
oc:  [
    [0x0 0x1 0x2 0x3] [0x0 1x0 2x0 3x0] [0x0 0x1 0x2 1x1]
    [0x1 1x0 1x1 2x1] [0x1 1x0 1x1 1x2] [0x0 1x0 1x1 2x0]
    [0x0 0x1 0x2 1x0] [0x0 0x1 1x1 2x1] [0x2 1x0 1x1 1x2]
    [0x0 1x0 2x0 2x1] [0x0 0x1 0x2 1x2] [0x1 1x1 2x0 2x1]
    [0x0 1x0 1x1 1x2] [0x0 0x1 1x0 2x0]    [0x0 0x1 1x1 1x2]
    [0x1 1x0 1x1 2x0] [0x1 0x2 1x0 1x1] [0x0 1x0 1x1 2x1]
    [0x0 0x1 1x0 1x1]
]
```

I remove the "top" and "edge" blocks, and replace all code references to them with "oc".

Now there's another bug I need to fix. Sometimes when I press a key, the following error occurs:

```
** Script Error: Invalid argument: [45
** Where: to-integer            |
** Near: forall arg [change arg to-integer first arg]
arg: to-pair
```

The code referenced is not part of any code I've written. It seems to be related to keystroke input because it only happens when I press one of the game control keys. Since I'm not sure what's creating the error (maybe it's related to the timing of keystrokes, or perhaps it has to do with a key release), I make an educated guess and figure that the following line, which waits for keystrokes, is where it's occurring:

```
if not none? wait/all [keys :00:00.30] [...]
```

I wrap that whole thing in an error check:

```
if not error? try [if not none? wait/all [keys :00:00.30] [...]]
```

And, hmmm ... that doesn't work. So instead of guessing, I work methodically to check each of the other main sections of the program. Every section gets wrapped in an "error? try" routine, and I also put in an "if" conditional stucture to print out a numbered error message whenever an error occurs. I find that the error is first occurring here:

```
do compose/deep [prin tui [at (pos)] print tui shape/(r)]
```

Wrapped in the error test, that section looks like this:

```
if error? try [
    do compose/deep [
        prin tui [at (pos)] print tui shape/(r)
    ]
] [print "er1"]
```

I'm curious about what's causing the error, so I dig a little deeper. This time I have the error check print out the variables contained in the code:

```
if error? try [
    do compose/deep [
        prin tui [at (pos)] print tui shape/(r)
    ]
] [print rejoin [pos " " r]]
```

Nothing seems to be amiss. Every time the error occurs, the variables show a correct coordinate and shape number. So, for now I'll simply leave the error check in place, removing the printout. This will keep the game moving along whenever the ghostly error occurs. I'll need to post a message to the REBOL mailing list to see if anyone knows why the error is occurring. For the time being, the following error handler fixes the issue:

```
if error? try [
    do compose/deep [
        prin tui [at (pos)] print tui shape/(r)
    ]
] []
```

It turns out that I need to do the same thing for all the other similar occurences of code that print a shape to the screen:

```
if error? try [
    do compose/deep [
        prin tui [at (pos)] print tui shape/(old-r + 19)
    ]
] []

if error? try [
```

```
    do compose/deep [
        prin tui [at (pos)] print tui shape/(old-r)
    ]
] []
```

With all the known bugs controlled, I can move on to implementing the last parts of the game design. We need to check if the top row of the playing area is reached. If any shape stops moving at this ceiling row, end the game. This needs to be done any time a piece reaches it's final resting place, so I put it immediately after the main "for" loop in the program outline (so that it's evaluated immediately after the stopping code is executed):

```
if (first pos) < 2 [
    prin tui [at 35x0]
    print "Game Over"
    halt
]
```

Finally, to erase the bottom line of shapes every time a row is filled in horizontally, we're going to have to redraw the playing field entirely. The "floor" block contains all the information needed to rebuild the current state of the playing field (all the positions at which a character is currently printed). Here's an outline and some pseudo code to think through what needs to be done:

1. Every time a shape stops moving, check to see if any row of the floor is full (i.e., there's one character printed in every column). I can use a for loop and a find function to perform that check on the floor block. (I'll start things off by just checking the bottom row).
2. If any row is full (for now, just the bottom row), remove that row of characters from the floor block. Use a remove-each loop to remove any coordinates that have y positions in the relevant row from the floor block.
3. Move all of the other characters above the relevant row down one row. Add one y position to all the other coordinates in the floor block which are above the relevant row. Use a foreach loop to go through each coordinate in the block and add 1x0. To replace the old floor block with the new one, first create a temporary block made up of the new floor block coordinates, then copy it back to the floor block once it's complete.
4. Erase the current screen, print the static background, and then reprint a new playing field using the refreshed block of floor coordinates. We can accomplish this easily using a foreach loop and TUI to print the characters at each coordinate in the list.

```
; #1:

line-is-full: true
for colmn 5 32 1 [
    each-coord: to-pair rejoin [29 "x" colmn]
    if not find floor each-coord [
        line-is-full: false
        break
    ]
]

; #2:

if line-is-full = true [
    remove-each cor floor [(first cor) = 29]

; #3:

    new-floor: copy []
    foreach cords floor [
        append new-floor (cords + 1x0)
    ]
```

```
    floor: copy new-floor

; #4:

    prin tui [clear]
    a-line: copy [] loop 28 [append a-line " "]
    a-line: rejoin ["   |"  to-string a-line "|"]
    loop 30 [print a-line]
    prin "   " loop 30 [prin "+"] print ""
    foreach was-here floor [
        if not ((first was-here) = 30) [
            prin tui compose [at (was-here)]
            prin "#"
        ]
    ]
]
```

At this point, I realize that I've made some logic errors in how the floor block and the stopping routine are structured. As it stands, when the screen is refreshed, the bottom row of the block (row 30) needs to be erased so that all the characters in row 29 can fall down one position. But if row 30 is erased, then the bottom of the floor disappears. As it turns out, row 31 should actually be treated as the bottom row, and all the characters should stop at 1x0 position higher then any character in the floor.

I make the required changes to the coordinates in the floor block (change all the y positions from 30 to 31). I also change the "new-floor-coord" variable in the stopping routine, and adjust the code above so that characters below line 30 are not printed. Additionally, the entire section above gets wrapped in a "for" loop to check if each row 1-30 is full. In the code above, I only checked if the bottom line was full - the number 29 referred to the row. I replace that number with the "row" variable created in the for loop. And with that, the last requirements of my original game outline are satisfied and an initial version of "Textris" is in working order. Here's the code:

```
REBOL [Title: "Textris"]

tui: func [
    {Cursor positioning dialect (iho)}
    [catch]
    commands [block!]
    /local screen-size string arg cnt cmd c err
][
    screen-size: (
        c: open/binary/no-wait [scheme: 'console]
        prin "^(1B)[7n"
        arg: next next to-string copy c
        close c
        arg: parse/all arg ";R"
        forall arg [change arg to-integer first arg]
        arg: to-pair head arg
    )
    string: copy ""
    cmd: func [s][join "^(1B)[" s]
    if error? set/any 'err try [
        commands: compose bind commands 'screen-size ][
        throw err
    ]
    arg: parse commands [
        any [
            'direct set arg string! (append string arg) |
            'home  (append string cmd "H") |
            'kill  (append string cmd "K") |
            'clear (append string cmd "J") |
```

```
            'up    set arg integer! (append string cmd [
                arg "A"]) |
            'down  set arg integer! (append string cmd [
                arg "B"]) |
            'right set arg integer! (append string cmd [
                arg "C"]) |
            'left  set arg integer! (append string cmd [
                arg "D"]) |
            'at   set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
            'del   set arg integer! (append string cmd [
                arg "P"]) |
            'space set arg integer! (append string cmd [
                arg "@"]) |
            'move  set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
            set cnt integer! set arg string! (
                append string head insert/dup copy "" arg cnt
            ) |
            set arg string! (append string arg)
        ]
        end
    ]
    if not arg [throw make error! "Unable to parse block"]
    string
]

shape: [
    ["####"]
    ["#" down 1 left 1 "#" down 1 left 1 "#" down 1 left 1 "#"]
    ["###" down 1 left 2 "#"]
    [right 1 "#" down 1 left 2 "##" down 1 left 1 "#"]
    [right 1 "#" down 1 left 2 "###"]
    ["#" down 1 left 1 "##" down 1 left 2 "#"]
    ["###" down 1 left 3 "#"]
    ["##" down 1 left 1 "#" down 1 left 1 "#"]
    [right 2 "#" down 1 left 3 "###"]
    ["#" down 1 left 1 "#" down 1 left 1 "##"]
    ["###" down 1 left 1 "#"]
    [right 1 "#" down 1 left 1 "#" down 1 left 2 "##"]
    ["#" down 1 left 1 "###"]
    ["##" down 1 left 2 "#" down 1 left 1 "#"]
    ["##" down 1 left 1 "##"]
    [right 1 "#" down 1 left 2 "##" down 1 left 2 "#"]
    [right 1 "##" down 1 left 3 "##"]
    ["#" down 1 left 1 "##" down 1 left 1 "#"]
    ["##" down 1 left 2 "##"]
    ;
    ["    "]
    [" " down 1 left 1 " " down 1 left 1 " " down 1 left 1 " "]
    ["   " down 1 left 2 " "]
    [right 1 " " down 1 left 2 "  " down 1 left 1 " "]
    [right 1 " " down 1 left 2 "   "]
    [" " down 1 left 1 "  " down 1 left 2 " "]
    ["   " down 1 left 3 " "]
    ["  " down 1 left 1 " " down 1 left 1 " "]
    [right 2 " " down 1 left 3 "   "]
    [" " down 1 left 1 " " down 1 left 1 "  "]
    ["   " down 1 left 1 " "]
    [right 1 " " down 1 left 1 " " down 1 left 2 "  "]
    [" " down 1 left 1 "   "]
    ["  " down 1 left 2 " " down 1 left 1 " "]
    ["  " down 1 left 1 "  "]
```

```
    [right 1 " " down 1 left 2 "  " down 1 left 2 " "]
    [right 1 "  " down 1 left 3 "  "]
    [" " down 1 left 1 "  " down 1 left 1 " "]
    ["  " down 1 left 2 "  "]
]

floor:  [
    31x5 31x6 31x7 31x8 31x9 31x10 31x11 31x12 31x13 31x14 31x15
    31x16 31x17 31x18 31x19 31x20 31x21 31x22 31x23 31x24 31x25
    31x26 31x27 31x28 31x29 31x30 31x31 31x32
]

oc:  [
    [0x0 0x1 0x2 0x3] [0x0 1x0 2x0 3x0] [0x0 0x1 0x2 1x1]
    [0x1 1x0 1x1 2x1] [0x1 1x0 1x1 1x2] [0x0 1x0 1x1 2x0]
    [0x0 0x1 0x2 1x0] [0x0 0x1 1x1 2x1] [0x2 1x0 1x1 1x2]
    [0x0 1x0 2x0 2x1] [0x0 0x1 0x2 1x2] [0x1 1x1 2x0 2x1]
    [0x0 1x0 1x1 1x2] [0x0 0x1 1x0 2x0] [0x0 0x1 1x1 1x2]
    [0x1 1x0 1x1 2x0] [0x1 0x2 1x0 1x1] [0x0 1x0 1x1 2x1]
    [0x0 0x1 1x0 1x1]
]

width: [4 1 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 2]

a-line: copy [] loop 28 [append a-line " "]
a-line: rejoin ["   |"  to-string a-line "|"]
loop 30 [print a-line] prin "   " loop 30 [prin "+"] print ""

keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 18
    for i 1 30 1 [
        pos: to-pair rejoin [i "x" xpos]
        if error? try [
            do compose/deep [
                prin tui [at (pos)] print tui shape/(r)
            ]
        ] []
        old-r: r
        old-xpos: xpos
        if not none? wait/all [keys :00:00.30] [
            keystroke: to-string to-char to-integer copy keys
            switch/default keystroke [
                "k" [if (xpos > 5) [
                        xpos: xpos - 1
                    ]]
                "l" [if (xpos < (33 - compose width/(r))) [
                        xpos: xpos + 1
                    ]]
                "o" [if (xpos < (33 - compose width/(r)))  [
                        switch to-string r [
                            "1" [r: 2]
                            "2" [r: 1]
                            "3" [r: 6]
                            "4" [r: 3]
                            "5" [r: 4]
                            "6" [r: 5]
                            "7" [r: 10]
                            "8" [r: 7]
                            "9" [r: 8]
                            "10" [r: 9]
```

```
                        "11" [r: 14]
                        "12" [r: 11]
                        "13" [r: 12]
                        "14" [r: 13]
                        "15" [r: 16]
                        "16" [r: 15]
                        "17" [r: 18]
                        "18" [r: 17]
                        "19" [r: 19]
                    ]
                ]
            ]
        ] []
    ]
    if error? try [
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r + 19)
        ]
    ] []
    stop: false
    foreach po compose oc/(r) [
        foreach coord floor [
            floor-y: to-integer first coord
            floor-x: to-integer second coord
            oc-y:  i + to-integer first po
            oc-x:  xpos + to-integer second po
            if (oc-y = (floor-y - 1)) and (floor-x = oc-x) [
                stop-shape-num: r
                stop: true
                break
            ]
        ]
    ]
    foreach po compose oc/(old-r) [
        foreach coord floor [
            floor-y: to-integer first coord
            floor-x: to-integer second coord
            oc-y:  i + to-integer first po
            oc-x:  old-xpos + to-integer second po
            if (oc-y = (floor-y - 1)) and (floor-x = oc-x) [
                stop-shape-num: old-r
                stop: true
                break
            ]
        ]
    ]
    if stop = true [
        left-col: second pos
        width-of-shape: length? compose oc/(stop-shape-num)
        right-col: left-col + width-of-shape - 1
        counter: 1
        for current-column left-col right-col 1 [
            add-coord: compose oc/(stop-shape-num)/(counter)
            new-floor-coord: (pos + add-coord)
            append floor new-floor-coord
            counter: counter + 1
        ]
        break
    ]
]
if (first pos) < 2 [
    prin tui [at 33x0]
    print "GAME OVER!!!"
```

```
        halt
    ]
    if error? try [
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r)
        ]
    ] []
    for row 1 30 1 [
        line-is-full: true
        for colmn 5 32 1 [
            each-coord: to-pair rejoin [row "x" colmn]
            if not find floor each-coord [
                line-is-full: false
                break
            ]
        ]
        if line-is-full = true [
            remove-each cor floor [(first cor) = row]
            new-floor: copy [
                31x5 31x6 31x7 31x8 31x9 31x10 31x11 31x12 31x13
                31x14 31x15 31x16 31x17 31x18 31x19 31x20 31x21
                31x22 31x23 31x24 31x25 31x26 31x27 31x28 31x29
                31x30 31x31 31x32
            ]
            foreach cords floor [
                either ((first cords) < row) [
                    append new-floor (cords + 1x0)
                ][
                    append new-floor cords
                ]
            ]
            floor: copy unique new-floor
            prin tui [clear]
            a-line: copy [] loop 28 [append a-line " "]
            a-line: rejoin ["   |"  to-string a-line "|"]
            loop 30 [print a-line]
            prin "   " loop 30 [prin "+"] print ""
            foreach was-here floor [
                if not ((first was-here) = 31) [
                    prin tui compose [at (was-here)]
                    prin "#"
                ]
            ]
        ]
    ]
]
```

Now that the program is working to my original specs, I want to make it look a bit spiffier. First of all, the playing area looks too wide and tall. I check Rebtris, and it's only 10 columns wide by 20 rows tall. I like that look and feel, so I adjust the floor block, the code that draws the static backdrop, and all computations related to the right boundaries of the playing field and the number of rows, to reflect that change.

I also want to print out a "Textris" title header, some keyboard instructions, and a score header. Tui allows me to print this text to the right of the playing field where I want it:

```
print tui [
    at 4x21 "TEXTRIS" at 5x21 "-------"
    at 7x20 "'K' = left" at 8x20 "'L' = right"
    at 9x20 "'O' = spin" at 11x21 "Score:"
]
```

Keeping track of the score is simple. When the program starts, a "score" variable is created and set to 0 ("score: 0"). Every time a piece stops falling, 10 points are added to the score. That number is printed beneath the score header (notice that the score number must first be converted to a string, in order to be printed by tui):

```
score: score + 10
print tui compose [at 13x21 (to-string score)]
```

Every time a row is filled in, 1000 points are added to the score. When the screen if redrawn to reflect the newly erased row, the tui code that prints the backdrop also prints out the updated score:

```
print tui compose [
    at 4x21 "TEXTRIS" at 5x21 "-------"
    at 7x20 "'K' = left" at 8x20 "'L' = right"
    at 9x20 "'O' = spin" at 11x21 "Score:"
    at 13x21 (to-string score)
]
```

Next, I want to add a pause key. This will fit in the switch structure that watches for keystrokes. Whenever the "P" key is pressed, print a message indicating that the game has been paused. Use an "ask" action to wait for input, and then print two blank lines to erase the pause message and any errant characters that the user may type in before hitting the [Enter] key:

```
"p" [
    print tui [
        at 23x0 "Press [Enter] to continue"
    ]
    ask ""
    print tui [
        at 24x0 "                                   "
        at 23x0 "                                   "
    ]
]
```

After posting some of this code to the REBOL mail list, another bug has become obvious. If the insert key or the arrow keys are pressed during game play, the game crashes. The following code produces a "** Math Error: Math or number overflow" when those keys are evaluated:

```
keystroke: to-string to-char to-integer copy keys
```

To fix that, I create my own error check. The keys codes for the arrow keys are #{1B5B41}, #{1B5B42}, #{1B5B43}, #{1B5B44}, and #{1B5B327E}. I check to see if they've been pressed first. If not, run the code above:

```
now-key: copy keys
if not (
    find [
        #{1B5B41} #{1B5B42} #{1B5B43}
        #{1B5B44} #{1B5B327E}
    ] (now-key)
) [keystroke: to-string to-char to-integer now-key]
```

That works, but a message to the list by Gabrielle Santilli creates a simpler solution. It turns out that I should have looked at the console port format a bit more carefully. All that's needed to get the keystroke is:

```
keystroke: to-string copy keys
```

And that does not produce errors for any entered keys.

I added all the above code to the program, and then tested everything. In doing so, I made an interesting discovery - it turns out that the code which produced the ghostly key input error in the shape printing routines is in a section of the TUI dialect that enables one to check for screen size. I think the error has something to do with the fact that I'm "compose"ing the results - not sure, but it doesn't matter. Since I'm not using that function, I simply remove it from the code. While I'm at it, I remove all the other parts of the TUI dialect that I'm not using. It turns out that all I need is:

```
tui: func [commands [block!]] [
    string: copy ""
    cmd: func [s][join "^(1B)[" s]
    arg: parse commands [
        any [
            'clear (append string cmd "J") |
            'up    set arg integer! (append string cmd [
                arg "A"]) |
            'down  set arg integer! (append string cmd [
                arg "B"]) |
            'right set arg integer! (append string cmd [
                arg "C"]) |
            'left  set arg integer! (append string cmd [
                arg "D"]) |
            'at   set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
            set arg string! (append string arg)
        ]
        end
    ]
    string
]
```

With that error gone, I can remove all the error checking routines in the program (they were causing some additional problems). Now Textris feels like a reasonably complete program. Here's the final code:

```
REBOL [Title: "Textris"]

tui: func [commands [block!]] [
    string: copy ""
    cmd: func [s][join "^(1B)[" s]
    arg: parse commands [
        any [
            'clear (append string cmd "J") |
            'up    set arg integer! (append string cmd [
                arg "A"]) |
            'down  set arg integer! (append string cmd [
                arg "B"]) |
            'right set arg integer! (append string cmd [
                arg "C"]) |
            'left  set arg integer! (append string cmd [
                arg "D"]) |
            'at   set arg pair! (append string cmd [
                arg/x ";" arg/y "H" ]) |
            set arg string! (append string arg)
        ]
```

```
        end
    ]
    string
]

shape: [
    ["####"]
    ["#" down 1 left 1 "#" down 1 left 1 "#" down 1 left 1 "#"]
    ["###" down 1 left 2 "#"]
    [right 1 "#" down 1 left 2 "##" down 1 left 1 "#"]
    [right 1 "#" down 1 left 2 "###"]
    ["#" down 1 left 1 "##" down 1 left 2 "#"]
    ["###" down 1 left 3 "#"]
    ["##" down 1 left 1 "#" down 1 left 1 "#"]
    [right 2 "#" down 1 left 3 "###"]
    ["#" down 1 left 1 "#" down 1 left 1 "##"]
    ["###" down 1 left 1 "#"]
    [right 1 "#" down 1 left 1 "#" down 1 left 2 "##"]
    ["#" down 1 left 1 "###"]
    ["##" down 1 left 2 "#" down 1 left 1 "#"]
    ["##" down 1 left 1 "##"]
    [right 1 "#" down 1 left 2 "##" down 1 left 2 "#"]
    [right 1 "##" down 1 left 3 "##"]
    ["#" down 1 left 1 "##" down 1 left 1 "#"]
    ["##" down 1 left 2 "##"]
    ;
    ["    "]
    [" " down 1 left 1 " " down 1 left 1 " " down 1 left 1 " "]
    ["   " down 1 left 2 " "]
    [right 1 " " down 1 left 2 "  " down 1 left 1 " "]
    [right 1 " " down 1 left 2 "   "]
    [" " down 1 left 1 "  " down 1 left 2 " "]
    ["   " down 1 left 3 " "]
    ["  " down 1 left 1 " " down 1 left 1 " "]
    [right 2 " " down 1 left 3 "   "]
    [" " down 1 left 1 " " down 1 left 1 "  "]
    ["   " down 1 left 1 " "]
    [right 1 " " down 1 left 1 " " down 1 left 2 "  "]
    [" " down 1 left 1 "   "]
    ["  " down 1 left 2 " " down 1 left 1 " "]
    ["  " down 1 left 1 "  "]
    [right 1 " " down 1 left 2 "  " down 1 left 2 " "]
    [right 1 "  " down 1 left 3 "  "]
    [" " down 1 left 1 "  " down 1 left 1 " "]
    ["  " down 1 left 2 "  "]
]
floor:  [
    21x5 21x6 21x7 21x8 21x9 21x10 21x11 21x12 21x13 21x14 21x15
]
oc:  [
    [0x0 0x1 0x2 0x3] [0x0 1x0 2x0 3x0] [0x0 0x1 0x2 1x1]
    [0x1 1x0 1x1 2x1] [0x1 1x0 1x1 1x2] [0x0 1x0 1x1 2x0]
    [0x0 0x1 0x2 1x0] [0x0 0x1 1x1 2x1] [0x2 1x0 1x1 1x2]
    [0x0 1x0 2x0 2x1] [0x0 0x1 0x2 1x2] [0x1 1x1 2x0 2x1]
    [0x0 1x0 1x1 1x2] [0x0 0x1 1x0 2x0] [0x0 0x1 1x1 1x2]
    [0x1 1x0 1x1 2x0] [0x1 0x2 1x0 1x1] [0x0 1x0 1x1 2x1]
    [0x0 0x1 1x0 1x1]
]
width: [4 1 3 2 3 2 3 2 3 2 3 2 3 2 3 2 2]
score: 0

prin tui [clear]
a-line: copy [] loop 11 [append a-line " "]
```

```
a-line: rejoin ["   |"  to-string a-line "|"]
loop 20 [print a-line] prin "   " loop 13 [prin "+"] print ""
print tui compose [
    at 4x21 "TEXTRIS" at 5x21 "-------"
    at 7x20 "Use arrow keys" at 8x20 "to move/spin."
    at 10x20 "'P' = pause"
    at 13x20 "SCORE:  " (to-string score)
]

keys: open/binary/no-wait [scheme: 'console]
forever [
    random/seed now
    r: random 19
    xpos: 9
    for i 1 20 1 [
        pos: to-pair rejoin [i "x" xpos]
        do compose/deep [prin tui [at (pos)] print tui shape/(r)]
        old-r: r
        old-xpos: xpos
        if not none? wait/all [keys :00:00.30] [
            switch/default to-string copy keys [
                "p" [
                    print tui [
                        at 23x0 "Press [Enter] to continue"
                    ]
                    ask ""
                    print tui [
                        at 24x0 "                                  "
                        at 23x0 "                                  "
                    ]
                ]
                "^[[D" [if (xpos > 5) [
                        xpos: xpos - 1
                ]]
                "^[[C" [if (xpos < (16 - compose width/(r))) [
                        xpos: xpos + 1
                ]]
                "^[[A" [if (xpos < (16 - compose width/(r)))  [
                        switch to-string r [
                            "1" [r: 2]
                            "2" [r: 1]
                            "3" [r: 6]
                            "4" [r: 3]
                            "5" [r: 4]
                            "6" [r: 5]
                            "7" [r: 10]
                            "8" [r: 7]
                            "9" [r: 8]
                            "10" [r: 9]
                            "11" [r: 14]
                            "12" [r: 11]
                            "13" [r: 12]
                            "14" [r: 13]
                            "15" [r: 16]
                            "16" [r: 15]
                            "17" [r: 18]
                            "18" [r: 17]
                            "19" [r: 19]
                        ]
                    ]
                ]
            ] []
        ]
```

```
        do compose/deep [
            prin tui [at (pos)] print tui shape/(old-r + 19)
        ]
        stop: false
        foreach po compose oc/(r) [
            foreach coord floor [
                floor-y: to-integer first coord
                floor-x: to-integer second coord
                oc-y:  i + to-integer first po
                oc-x:  xpos + to-integer second po
                if (oc-y = (floor-y - 1)) and (floor-x = oc-x) [
                    stop-shape-num: r
                    stop: true
                    break
                ]
            ]
        ]
        foreach po compose oc/(old-r) [
            foreach coord floor [
                floor-y: to-integer first coord
                floor-x: to-integer second coord
                oc-y:  i + to-integer first po
                oc-x:  old-xpos + to-integer second po
                if (oc-y = (floor-y - 1)) and (floor-x = oc-x) [
                    stop-shape-num: old-r
                    stop: true
                    break
                ]
            ]
        ]
        if stop = true [
            left-col: second pos
            width-of-shape: length? compose oc/(stop-shape-num)
            right-col: left-col + width-of-shape - 1
            counter: 1
            for current-column left-col right-col 1 [
                add-coord: compose oc/(stop-shape-num)/(counter)
                new-floor-coord: (pos + add-coord)
                append floor new-floor-coord
                counter: counter + 1
            ]
            break
        ]
    ]
    do compose/deep [prin tui [at (pos)] print tui shape/(old-r)]
    if (first pos) < 2 [
        prin tui [at 23x0]
        print "   GAME OVER!!!^/^/"
        halt
    ]
    score: score + 10
    print tui compose [at 13x28 (to-string score)]
    for row 1 20 1 [
        line-is-full: true
        for colmn 5 15 1 [
            each-coord: to-pair rejoin [row "x" colmn]
            if not find floor each-coord [
                line-is-full: false
                break
            ]
        ]
        if line-is-full = true [
            remove-each cor floor [(first cor) = row]
```

```
            new-floor: copy [
                21x5 21x6 21x7 21x8 21x9 21x10 21x11 21x12 21x13
                21x14 21x15
            ]
            foreach cords floor [
                either ((first cords) < row) [
                    append new-floor (cords + 1x0)
                ][
                    append new-floor cords
                ]
            ]
            floor: copy unique new-floor
            score: score + 1000
            prin tui [clear]
            loop 20 [print a-line]
            prin "   " loop 13 [prin "+"] print ""
            print tui compose [
                at 4x21 "TEXTRIS" at 5x21 "-------"
                at 7x20 "Use arrow keys" at 8x20 "to move/spin."
                at 10x20 "'P' = pause"
                at 13x20 "SCORE:  " (to-string score)
            ]
            foreach was-here floor [
                if not ((first was-here) = 21) [
                    prin tui compose [at (was-here)]
                    prin "#"
                ]
            ]
        ]
    ]
]
```

Here's a quick synopsis of the program:

- The TUI dialect is defined.
- The "shape" block, containing the TUI instructions for drawing each shape is defined.
- The "floor", "oc", and "width" coordinate blocks are defined. The "score" variable is also defined.
- The backdrop characters (left, right, and bottom barriers), instructions, headers, and score are printed.
- A forever loop runs the main actions of the program. The subsections of that loop are:
    - A shape is printed on the screen.
    - User keystrokes are watched for.
    - A switch structure decides what to do with entered keystrokes (right arrow = move right, left arrow = move left, up arrow = rotate shape, p = pause).
    - Another switch structure determines which shape to print when the current shape is rotated.
    - The currently printed shape is erased.
    - Two foreach loops check whether the current shape has reached a position at which it should stop falling.
    - If the piece has reached a stopping point, the coordinates occupied by the piece are added to the "floor" block.
    - The shape is printed at its final resting place.
    - If the current shape touches the ceiling, the game ends.
    - The score is updated.
    - If any rows have been completely filled in, their coordinates are removed from the floor block, the coordinates of all other characters are moved down a row, and the screen is reprinted with the new floor coordinates and the new score.
    - The forever loop continues.

If I'd been so thoughtful and organized as to write a structured outline like that in the beginning of the case study, things would've moved along more quickly. But any project is easier in retrospect ... I

just try to remember that building as detailed an outline as possible before writing any code always saves a great deal of work and confusion.

Now that the game satisfies my original intentions, I'll bring the case study to a close, but not without first putting together a to-do list of things to improve in the program. If you'd like to try implementing some of these changes, first figure out where in the outline they should go, write some pseudo code to get the job done, and then come up with REBOL code to satisfy those pseudo code expressions:

1. Save high scores to disk.
2. Add a way to incrementally increase the speed at which shapes drop. Do this every time a certain number of rows is cleared.
3. Add a "next piece" preview.
4. Look for a way to remove the cursor from the printout, so that it's not visible along the left side of the wall as the shapes fall.
5. Add sound. Play tones for each event that occurs, and play a background tune while the game is running.
6. Rewrite the entire program using GUI techniques, instead of console text characters and TUI.

Looking at my coding process in retrospect, I should note some criticisms. One element that annoyed me was a set of badly chosen variable names. I initially used "r", for example, to represent the current shape number because it was first used to represent a random number. "R" is not so descriptive, and it was hard to remember what "r" represented while I was coding. The same was true of "i", which became more important as the loop that dropped the shapes grew in complexity. I left those variables as they were in this case study so that the lines of code fit neatly onto this web page, but in my own coding I choose to use more descriptive variables. Doing that in general makes code more readable and easier to think through.

The Moral of the Story:

Whether or not you're interested in game programming, and despite the fact that the final product of this case study is a bland implementation of Tetris, some general understanding about coding can be gained from the thought process covered in this section. It's typical of any general coding project you'll encounter: start with a design concept, outline the main structure of the program you imagine, use pseudo code to guide you from the "what am I trying to do?" through the "how do I code it" stages, and refine the detail of your outline by testing and experimenting with small code chunks along the way.

In general, if you can't think through the process of "what am I trying to accomplish" in a structured way, then you won't be able to write the code to accomplish it. Once you've got a basic grasp of language concepts and syntax, you'll see that writing code just takes lots of creative organization and experimentation. Keep a language reference close at hand, and you can work out the syntax of virtually any code you need to write. That's only a matter of knowing which functions and constructs are available to solve your problems, and looking up the format for those you're not familiar with. The difficult part in any coding situation is mapping each small thought process to a data construct, conditional expression, looping routine, function definition, existing code module, word label, etc. For large projects, you'll typically need an outline because it's so easy to get lost in the minute coding details along the way. Start with a top down approach, conceive and design a flow chart/outline, and then flesh out the details of each section until you've got code written to solve each design concept. Once you become familiar with that process, experience will show that you can code solutions for virtually any problem you encounter.

You'll find that in many cases REBOL allows you to think directly in code more easily than you can with pseudo code. That's because REBOL's high level design is meant to be human readable and human "thinkable". Although many coding concepts in all computer languages are generally the same, most other languages are more overtly designed and constrained by legacy concepts derived from requirements about how computers operate. Some languages tend to require much more low level coding or coersing of disparate modules to fit together in order to make the conceptual design take shape in final code form. Other languages get you bogged down in thinking about higher level OOP constructs. A lack of universal data structures such as REBOL's block/series structure, a lack of built in native data types such as time, tuples, pairs, money, etc., and a less natural way of structuring functions, variables and module definitions (not using words and dialects in a natural language way), require unique and contrived constructs to be designed to manipulate data in each

individual program. In the most popular languages, program authors typically have to be more concerned about managing the rudimentary memory and cpu actions of the computer for everything that occurs in a program. That enables a greater measure of control over how a computer uses it's hardware resources, but it's very far from the way humans naturally think about solving real life situations. REBOL allows things to be done in a way of thinking that's closer to the outline stage. When you get used to writing REBOL code, you'll find that it saves a tremendous amount of time compared to other languages. Remember along the way that no matter what computer language(s) you learn, understanding how to think through the "what I am I trying to accomplish" outline is essential to writing code that accomplishes your task.

10.10 Case 10 - Scheduling Teachers, Part Two

After several months of using the teacher scheduling application described in the first case study, my business expanded, and the teaching staff grew. With the way things worked in my short initial program, I would have to create a new folder on the web site and compile a unique version of the program for each new teacher. This would require recompiling and uploading a new version, for each teacher, every time I alter the program. I wanted to make a multi-user version of the application to simplify setup and to save maintenance time. I also wanted to add some error checking and a simple password scheme to the existing program. To create a new version of the application, here's my concept in outline/pseudo code form:

1. Maintain the existing folder and file structure on the web site (http://website.com/teacher/name, schedule.txt, and index.php).
2. Add a file to the web site containing a list of current teachers and associated passwords. Put it outside the public_html folder, so that people can't download it without a password.
3. In the application, start by downloading that file from the website (using ftp).
4. Display a text list of teacher names from the downloaded file.
5. When a teacher name is selected, request a password from the user and check that it matches the associated password for the given teacher.
6. Append the teacher name to the http and ftp URLs, and run the program as before.
7. Add some error checking and backup routines every time the data is read or written locally, or on the web server. That way, no data is ever lost.
8. Compile the program and upload it to the web site. Point all links on the index.php pages to that single file. Now, any time I want to add a new teacher, all I need to do is add the new teacher name and password to the downloadable text file and copy a blank index.php and schedule.txt to a new folder on the web server. If I ever make additional changes to the program, I only need to recompile and upload that single program file.

To start things off, I created a text file called "teacherlist.txt" and stored it outside the public_html folder on the web server. It's formatted like this:

```
["mark" "markspassword"] ["ryan" "ryanspassword"]
["nick" "nickspassword"] ["peter" "peterspassword"]
["rudi" "rudispassword"] ["tom" "tomspassword"]
```

The first thing I do in the program is read the data:

```
teacherlist: load ftp://user:pass@website.com/teacherlist.txt
```

Next, display a list of the teachers. The first item in each block of teacherlist.txt is the teacher name. A foreach loop reads each of those names into a new block, and that block is displayed using a GUI text-list widget:

```
teachers: copy []
foreach teacher teacherlist [append teachers first teacher]
view center-face layout [
    text-list data teachers [folder: value unview]
]
```

Next, get the password from the user and use a foreach loop to look through the list, checking for a match in teacher names and passwords entered by the user (the first and second elements, respectively, in each block):

```
pass: request-pass/only
correct: false
```

```
foreach teacher teacherlist [
    if ((first teacher) = folder) and (pass = (second teacher)) [
        correct: true
    ]
]
if correct = false [alert "Incorrect password." quit]
```

I add the following line to the script, which keeps REBOL from terminating the script when the [Esc] key is pressed. That behavior is the default in the REBOL interpreter, and makes it easy for someone to just stop the script and view the teacherlist. (I'm not so concerned about security here, but I don't want passwords to be blatantly available):

```
system/console/break: false
```

Finally, I come up with an error message to be executed any time an Internet connection isn't available. It allows the user to read any of the recently backed up schedule.txt files so that the program is useful even if an Internet connection isn't available:

```
error-message: does [
    ans: request {Internet connection is not available.
        Would you like to see one of the recent local backups?}
    either ans = true [
        editor to-file request-file quit
    ][
        quit
    ]
]
```

I wrap all attempts to connect to the Internet in "error? try" routines, and duplicate the original backup routine from the initial program so that no data is ever lost. Here's the final code:

```
REBOL [title: "Lesson Scheduler"]

system/console/break: false
error-message: does [
    ans: request {Internet connection is not available.
        Would you like to see one of the recent local backups?}
    either ans = true [
        editor to-file request-file quit
    ][
        quit
    ]
]

if error? try [
    teacherlist: load ftp://user:pass@website.com/teacherlist.txt
][
    error-message
]
teachers: copy []
foreach teacher teacherlist [append teachers first teacher]
view center-face layout [
    text-list data teachers [folder: value unview]
]

pass: request-pass/only
correct: false
foreach teacher teacherlist [
```

```
    if ((first teacher) = folder) and (pass = (second teacher)) [
        correct: true
    ]
]
if correct = false [alert "Incorrect password." quit]

url: rejoin [http://website.com/teacher/ folder]
ftp-url: rejoin [
    ftp://user:pass@website.com/public_html/teacher/ folder
]

if error? try [
    write %schedule.txt read rejoin [url "/schedule.txt"]
][
    error-message
]

; backup (before changes are made):
cur-time: to-string replace/all to-string now/time ":" "-"
; local:
write to-file rejoin [
    folder "-schedule_" now/date "_" cur-time ".txt"
] read %schedule.txt
; online:
if error? try [
    write rejoin [
        ftp-url "/" now/date "_" cur-time
    ] read %schedule.txt
][
    error-message
]

editor %schedule.txt

; backup again (after changes are made):
cur-time: to-string replace/all to-string now/time ":" "-"
write to-file rejoin [
    folder "-schedule_" now/date "_" cur-time ".txt"
] read %schedule.txt
if error? try [
    write rejoin [
        ftp-url "/" now/date "_" cur-time
    ] read %schedule.txt
][
    alert "Internet connection not available while backing up."
]

; save to web site:
if error? try [
    write rejoin [ftp-url "/schedule.txt"] read %schedule.txt
][
    alert {Internet connection not available while updating web
    site.  Your schedule has NOT been saved online.}
    quit
]
browse url
```

With the new application complete, I wanted to create an additional cgi application for the web site to collectively display all available times in each of the teachers' schedules. This would help with scheduling because both students and management could instantly see a bird's eye view of all open appointment times, on a single web page. In order for that display to be viewable by the general public, I want the cgi app to strip out all personal data contained in the schedules. To create the cgi, I need to search each line of schedule text for "----". If a line contains the characters "----", that time is

available. Here's a pseudo code outline that I thought through as I organized the process:

1. Make a list of all the teacher pages. Store the links in a block.
2. Use a foreach loop to cycle through each page in the list. Read in the data on each page in line format, using another foreach loop.
3. For each line, use a find function to check whether the line contains the name of a day of the week, or the characters "----". If so, print the line, adding some additional formatting to separate days as headers. Also print each page link as a header separating each teacher's schedule in the printout.

First, I created the block of links:

```
page-list: [
    http://website.com/teacher/ryan
    http://website.com/teacher/mark
    http://website.com/teacher/nick
    http://website.com/teacher/peter
    http://website.com/teacher/tom
    http://website.com/teacher/rudi
]
```

For step 2, I created the foreach loop to read each page:

```
foreach page page-list [
    data: read/lines page
]
```

Inside that loop, I added the code to print out the teacher name and day headers, and the available times:

```
foreach page page-list [
    print newline
    print to-string page
    print ""
    data: read/lines page
    week: ["MONDAY" "TUESDAY" "WEDNESDAY" "THURSDAY" "FRIDAY"
        "SATURDAY" "SUNDAY"]
    foreach line data [
        foreach day week [
            if find line day [print "" print line print ""]
        ]
        if find line "----" [print line]
    ]
]
```

Now I've got a little command line application that does what I need:

```
REBOL []
page-list: [
    http://website.com/teacher/ryan
    http://website.com/teacher/mark
    http://website.com/teacher/nick
    http://website.com/teacher/peter
    http://website.com/teacher/tom
    http://website.com/teacher/rudi
]
foreach page page-list [
```

```
    print newline
    print to-string page
    print ""
    data: read/lines page
    week: ["MONDAY" "TUESDAY" "WEDNESDAY" "THURSDAY" "FRIDAY"
        "SATURDAY" "SUNDAY"]
    foreach line data [
        foreach day week [
            if find line day [print "" print line print ""]
        ]
        if find line "----" [print line]
    ]
]
```

Next, to the basic CGI framework provided earlier in this tutorial, I simply added the code above. The only real changes I needed to make were some added "< B R >"s (HTML line ends) to make the text display properly in the browser:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Available Appointment Times"</TITLE>]
print [</HEAD><BODY>]
page-list: [
    http://website.com/teacher/ryan
    http://website.com/teacher/mark
    http://website.com/teacher/nick
    http://website.com/teacher/peter
    http://website.com/teacher/tom
    http://website.com/teacher/rudi
]
foreach page page-list [
    print [<BR><BR>]
    print to-string page
    print [<BR>]
    data: read/lines page
    week: ["MONDAY" "TUESDAY" "WEDNESDAY" "THURSDAY" "FRIDAY"
        "SATURDAY" "SUNDAY"]
    foreach line data [
        foreach day week [
            if find line day [print [<BR>] print line print [<BR><BR>]]
        ]
        if find line "----" [print line print [<BR>]]
    ]
]
print [</BODY></HTML>]
```

As more teachers were added to the scheduling system, it became apparent that a CGI version of the editor would be helpful (for use on mobile phones, at work, and in other environments where installing an executable was problematic). For that, I simply used the password protected online text editor, found earlier in this tutorial:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Edit Schedule"</TITLE></HEAD><BODY>]

; submitted: decode-cgi system/options/cgi/query-string
```

```
; We can't use the above normal line to decode, because
; we're using the post method to submit data (because data
; from the textarea may get too big for the get method).
; Use the following function to process data from a post
; method instead:

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: decode-cgi read-cgi

; if schedule.txt has been edited and submitted:

if ((submitted/2 = "save") or (submitted/2 = "save")) [
    ; save newly edited schedule:
    write to-file rejoin ["./" submitted/6 "/schedule.txt"] submitted/4
    print ["Schedule Saved."]
    print rejoin [
       {<META HTTP-EQUIV="REFRESH" CONTENT="0;
           URL=http://website.com/folder/}
       submitted/6 {">}
    ]
    quit
]

; if user is just opening page (i.e., no data has been submitted
; yet), request user/pass:

if ((submitted/2 = none) or (submitted/4 = none)) [
    print [<strong>"W A R N I N G  -  "]
    print ["Private Server, Login Required:"</strong><BR><BR>]
    print [<FORM ACTION="./edit.cgi">]
    print [" Username: " <input type=text size="50" name="name"><BR><BR>]
    print [" Password: " <input type=text size="50" name="pass"><BR><BR>]
    print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
    print [</FORM>]
    quit
]

; check user/pass against those in teacherlist.txt,
; end if incorrect:

teacherlist: load %teacherlist.txt
folder: submitted/2
password: submitted/4
response: false
foreach teacher teacherlist [
    if ((first teacher) = folder) and (password = (second teacher)) [
        response: true
    ]
]
```

```
if response = false [print "Incorrect Username/Password." quit]

; if user/pass is ok, go on:

; backup (before changes are made):

cur-time: to-string replace/all to-string now/time ":" "-"
schedule_text: read to-file rejoin ["./" folder "/schedule.txt"]
write to-file rejoin [
    "./" folder "/" now/date "_" cur-time ".txt"] schedule_text

print [<strong>"Be sure to SUBMIT when done:"</strong><BR><BR>]
print [<FORM method="post" ACTION="./edit.cgi">]
print [<INPUT TYPE=hidden NAME=submit_confirm VALUE="save">]
print [<textarea cols="100" rows="15" name="contents">]
print [schedule_text]
print [</textarea><BR><BR>]
print rejoin [{<INPUT TYPE=hidden NAME=folder VALUE="} folder {">}]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</BODY></HTML>]
```

Now there's a way for all the teachers to edit their schedules. I can add a new teacher to the system in about 5 seconds (just create a new directory on the server and copy blank schedule.txt and index.php files). Anyone involved in scheduling can make changes online, regardless of location or computer type, and everyone stays synchronized. In addition, anyone can get an instant bird's eye view of all available appointment times - this helps tremendously when scheduling new students and maintaining daily activities.

10.11 Case 11 - An Online Member Page CGI Program

One of my friends wanted to create an online member database for a local club. He wanted members to be able to sign up and add their contact information, upload photos, and add info about themselves. He was tired of manually making changes to the members' pages, and wanted users to be able to add, edit, and delete their own information. He wanted basic password enabled access so that users could only edit their own information, and he wanted a back end utility that allowed him to make changes as administrator, and which automatically saved each successive change to the database, so that no data could ever be lost. He also wanted users automatically emailed their password, in case they forgot.

Here was my basic thought process and plan of attack:

1. This will be an online system (a web site), so the user interface will be a set of HTML pages that display each user's information, as well as a set of HTML forms for users to enter information. We decided to have the page display the following fields: Name, Address, Phone, Email, Age, Language, Height, Date the user was added, and an uploaded photo.
2. The data will be stored in some sort of online database. Since this is a small group with only a few users, I decided to create a simple flat file database - just a text file filled with blocks of REBOL data, one block per user, stored on the web server.
3. The page that pulls the info from the database and displays it in the above HTML will basically be a REBOL CGI application that runs a "foreach" loop to print each of the entries in the above HTML format. The pages where the users enter their information will be forms that submit the information to a REBOL CGI that appends it to the database text file. The pages where the users edit their information will be forms that display the information currently in the selected entry, without the password. When the user submits the new password and updated info, the CGI checks that the submitted password matches the existing password for that entry, and then replaces the old block with the new one, in the database text file. The code for emailing the user a forgotten password and for automatically backing up data will also be put here.
4. An image upload/update page also needs to created. This will be an HTML form that accepts a local image file on the user's computer, submits that file to the CGI, which in turn writes that binary data to a directory on the web server, creates an HTML image link to it, and adds that link to the appropriate user entry in the database text file.
5. The back end will simply be the password protected text editor explored in case study #8, with links to all the backup text files, for easy recovery (copy/paste) of lost data.

Here was the basic HTML layout I came up with for step 1. Each entry in the database will be displayed using this template:

```
<HR><BR> Date/Time: 23-Mar-2008/13:11:42-7:00

<A HREF="./index.cgi?function=">edit | </A>
<A HREF="./index.cgi?function=">delete</A>

<TABLE background="" border=0 cellPadding=2 cellSpacing=2
    width="100%"><TR>

<TD width = "600">
<BR>
<FONT FACE="Courier New, Courier, monospace">Name:          </FONT>
<STRONG>The User Name Goes Here</STRONG>
<BR>
<FONT FACE="Courier New, Courier, monospace">Address:       </FONT>
<STRONG>The Address Goes Here</STRONG>
<BR>
<FONT FACE="Courier New, Courier, monospace">Phone:         </FONT>
<STRONG>The Phone</STRONG>
<BR>
<FONT FACE="Courier New, Courier, monospace">Email:         </FONT>
<STRONG>The Email</STRONG>
```

```
<BR>
<FONT FACE="Courier New, Courier, monospace">Age:           </FONT>
<STRONG>The Age</STRONG>
<BR>
<FONT FACE="Courier New, Courier, monospace">Language:      </FONT>
<STRONG>The Language</STRONG>
<BR>
<FONT FACE="Courier New, Courier, monospace">Height:        </FONT>
<STRONG>The Height</STRONG>
<BR><BR>
</TD>

<TD width="170" valign="center">
<A HREF="./default.jpg" target=_blank><IMG align=baseline alt=""
    border=0 hspace=0 src="./default.jpg" width="160" height="120">
</A>
</TD>

</TR></TABLE>

Some Additional Notes Go Here...

<BR><BR>
```

The database design for step 2 was even simpler to create. Here's an example of what each block looks like. Notice that each entry in the database is just a text string separated by spaces, for each field of info we want displayed on the member page. In the block, I added a link to a default image, in case the user didn't upload their own photo. This file was saved as %bb.db:

```
["Username" "19-Feb-2008/4:55:59-8:00" "1 Address St."
    "123-456-7890" "name@website.com" "40"
    {REBOL, C, C++, Python, PHP, Basic, AutoIt, others...}
    "6'" {I'm a nobody - just a test account.} "password"
    [
        {<a href = "./default.jpg" target=_blank>
        <IMG align=baseline alt="" border=0 hspace=0
        src="./default.jpg" width="160" height="120"></a>}
    ]
]

["Tester McUser" "22-Feb-2008/13:14:44-8:00" "1 Way Lane"
    "234-567-8910" "tester@website.com" "35" "REBOL"
    {5' 11"} "I'm just another test account." "password"
    [
        {<a href = "./files/photo.jpg" target=_blank>
        <IMG align=baseline alt="" border=0 hspace=0
        src="./files/photo.jpg" width="160" height="120"></a>}
    ]
]
```

At this point I could begin the work of step 3, creating a CGI program that prints the HTML page in step 1, with the above data. Here's a simple CGI script that simply prints the HTML design together with the entries from the database inserted:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print read %header.html
```

```
bbs: load %bb.db  ; load the database info

print [<center><table border=1 cellpadding=10 width=90%><tr><td>]
print {<TABLE background="" border=0 cellPadding=0 cellSpacing=0
    height="100%" width="100%"><tr><td width = "600">}
print [<hr>]
reverse bbs
foreach bb bbs [
    print [<BR>]
    print rejoin ["Date/Time: " bb/2]
    print "                   "
    print rejoin [{<a href="./index.cgi?function=">edit | </a>}]
    print rejoin [{<a href="./index.cgi?function=">delete</a>}]
    print "          "
    print {<TABLE background="" border=0 cellPadding=2
        cellSpacing=2 height="100%" width="100%"><tr>
        <td width = "600"><BR>}
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
        "Name:          </FONT><strong>" bb/1 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
         "Address:       </FONT><strong>" bb/3 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
        "Phone:         </FONT><strong>" bb/4 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
        "Email:         </FONT><strong>" bb/5 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
        "Age:           </FONT><strong>" bb/6 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
         "Language:      </FONT><strong>" bb/7 "</strong>"]
    print [<BR>]
    print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
         "Height:        </FONT><strong>" bb/8 "</strong>"]
    print [<BR><BR>]
    print </td>
    print {<td width = "170" valign = "center">}
    print bb/11 ; image link
    print {</td></tr></table>}
    print bb/9  ; "other information " text
    print [<BR><BR><HR>]
]
print [</td></tr></td></tr></td></tr></table>]
print [</td></tr></table></center>]
print read %footer.html
```

To that code, there were a number of features that I realized I should add. First, I wanted to munge email addresses so that they were less likely to get collected by spam bots. This line of code does the job well enough for my needs. It turns "name@address.com" into "name at address dot com":

```
(replace/all (replace bb/5 "@" " at ") "." " dot ")
```

I also wanted any http:// links in the "other information" section to be automatically linked. To do that, I used parse to search for "http://" and the ending space character, then wrapped that link in the required < A H R E F = ...> tags. Here's the code:

```
bb_temp: copy bb/9
```

```
bb_temp2: copy bb_temp
parse/all bb_temp [any [
    thru "http://" copy link to " " (replace bb_temp2
        (rejoin [{http://} link]) (rejoin [
            { <a href="} {http://} link
            {" target=_blank>http://} link {</a> }]))
        ]
    to end
]
```

Furthermore, I wanted to have line endings in the "other information" section automatically converted to HTML "< b r >"s, so that they display correctly on the web page. That's easy:

```
replace/all bb_temp newline "  <br>  "
```

My friend wanted a count displayed of the total number of members. That's also easy, with "length? bbs":

```
print rejoin [{<font size=5> Members:  () length? bbs ()</font></td>}]
```

I also added a "join now" link to the CGI page where users would be able to add themselves to the database (that page hasn't been created yet):

```
print {<td><a href="./add.cgi">Join Now</a></td></tr></table><BR>}
```

In order for users to edit/delete their info later, I needed to tag each displayed entry with a unique number to automatically select the appropriate block from the database. To do this, I added a counter variable to the foreach loop, and incremented it each time through the loop (counter: counter + 1). Then I replaced the generic edit and delete links in the code above . . .

```
print rejoin [{<a href="./index.cgi?function=">edit | </a>}]
print rejoin [{<a href="./index.cgi?function=">delete</a>}]
```

. . . with links that contain the counter, and which can be deciphered by a CGI program as "get" data:

```
print rejoin [
    {<a href="./index.cgi?function=edititemnumber&messagenumber=}
    counter {&Submit=Post+Message">edit | </a>}
]
print rejoin [
    {<a href="./index.cgi?function=deleteitemnumber&messagenumber=}
    counter {&Submit=Post+Message">delete</a>}
]
```

Here's the script, as it stands so far:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print read %header.html

bbs: load %bb.db
```

```
print [<center><table border=1 cellpadding=10 width=90%><tr><td>]

print-all: does [
    print {<TABLE background="" border=0 cellPadding=0 cellSpacing=0
        height="100%" width="100%"><tr><td width = "600">}
    print rejoin [{<font size=5> Members:  (} length? bbs {)</font></td>}]
    print {<td><a href="./add.cgi">Join Now</a></td></tr></table><BR>}
    print [<hr>]
    counter: 1
    reverse bbs
    foreach bb bbs [
        print [<BR>]
        if bb/1 <> "file uploaded" [
            print rejoin ["Date/Time: " bb/2]
            print "                    "
            print rejoin trim [
                {<a href=
                "./index.cgi?function=edititemnumber&messagenumber=}
                 counter
                {&Submit=Post+Message">edit | </a>}
            ]
            print rejoin trim [
                {<a href=
                "./index.cgi?function=deleteitemnumber&messagenumber=}
                counter
                {&Submit=Post+Message">delete</a>}
            ]
            print "         "
            print {
                    <TABLE background="" border=0 cellPadding=2
                    cellSpacing=2 height="100%" width="100%"><tr>
                    <td width = "600"><BR>
            }
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                "Name:          </FONT><strong>" bb/1 "</strong>"]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                 "Address:       </FONT><strong>" bb/3 "</strong>"]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                "Phone:         </FONT><strong>" bb/4 "</strong>"]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                "Email:         </FONT><strong>"
                (replace/all (replace bb/5 "@" " at ") "." " dot ")
                "</strong>"
            ]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                "Age:           </FONT><strong>" bb/6 "</strong>"]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                 "Language:      </FONT><strong>" bb/7 "</strong>"]
            print [<BR>]
            print rejoin [{<FONT FACE="Courier New, Courier, monospace">}
                 "Height:        </FONT><strong>" bb/8 "</strong>"]
            print [<BR><BR>]
        ]
        ; automatically convert line endings to HTML " <br>"
        bb_temp: copy bb/9
        replace/all bb_temp newline "  <br>  "
        bb_temp2: copy bb_temp
        ; automatically link any urls starting with http://
```

```
        append bb_temp " "
        parse/all bb_temp [any [
            thru "http://" copy link to " " (replace bb_temp2
                (rejoin [{http://} link]) (rejoin [
                    { <a href="} {http://} link
                    {" target=_blank>http://} link {</a> }]))
                ]
            to end
        ]
        print </td>
        print {<td width = "170" valign = "center">}
        print bb/11 ; image link
        print {</td></tr></table>}
        print bb_temp2
        print [<BR><BR><HR>]
        counter: counter + 1
    ]
    print [</td></tr></td></tr></td></tr></table>]
]
print-all
print [</td></tr></table></center>]
print read %footer.html
```

The page above was saved as index.cgi, and serves as the main display page for the site. In order to ensure that a fresh copy of that page is always viewed by visitors, I also created the following index.html page that simply refreshes the index.cgi page. By using that index.html page as the primary link (and by making that HTML file the default page for the web site), visitors always automatically see a refreshed view of the member page, with any changes/updates that have been made:

```
<html>
<head>
<title></title>
<META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.cgi">
</head>
<body bgcolor="#FFFFFF">
</body>
</html>
```

Next, I needed to create a form for users to enter their member information. This was saved as add.cgi. The form posts any submitted information back to index.cgi.

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print read %header.html

print [<center><table border=1 cellpadding=10 width=90%><tr><td>]
print [<font size=5>" Add New Member Information:"</font>]
print "         "
print "         "
print "         "
print [<hr>]
print [<FORM method="post" ACTION="./index.cgi">]
print [<br>" Your Name: " <br><input type=text size="60"
    name="username"><BR>]
print [<br>" Password (required to edit member info later): " <br>
    <input type=text size="60" name="password"><BR>]
print [<br>" Address: " <br><input type=text size="60" name="address">
```

```
    <BR>]
print [<br>" Phone: " <br><input type=text size="60" name="phone"><BR>]
print [<br>" Email: " <br><input type=text size="60" name="email"><BR>]
print [<br>" Age: " <br><input type=text size="60" name="age"><BR>]
print [<br>" Language: " <br><input type=text size="60" name="language">
    <BR>]
print [<br>" Height: " <br><input type=text size="60" name="height"><BR>
    <BR>]
print [" Other Information/Notes: " <br>]
print [<textarea name=otherinfo rows=5 cols=50></textarea><BR><BR>]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Post New Member Info">]
print [</FORM>]

print [</td></tr></table></center>]
print read %footer.html
```

I integrated the following code into index.cgi, to read and add the info from the above form to the database:

```
; here's the default code used to read any data from an HTML form:

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]
submitted: decode-cgi read-cgi

; make sure at least a user name and password was entered:

if submitted/2 <> none [
    if (submitted/2 = "") or (submitted/4 = "") [
    print {
        <strong>You must include at least
         a name and password.</strong>
        <br><br>Press the [BACK] button
        in your browser to try again.
    }
    print [</td></tr></table></center>]
    print read %footer.html
    halt
]

; now create a new entry block to add to the database:

entry: copy []
append entry submitted/2     ; name
; the time on the server is 3 hours different then our local time:
append entry to-string (now + 3:00)
append entry submitted/6      ; address
append entry submitted/8      ; phone
append entry submitted/10     ; email
```

```
append entry submitted/12      ; age
append entry submitted/14      ; language
append entry submitted/16      ; height
append entry submitted/18      ; other info
append entry submitted/4       ; password
append/only entry [
    {<a href = "./default.jpg" target=_blank>
    <IMG align=baseline alt="" border=0 hspace=0 src="./default.jpg"
    width="160" height="120"></a>}
]

; append the new entry to the database, and notify the user:

append/only tail bbs entry
save %bb.db bbs
print {<strong>New Member Added.</strong>
    Click "Edit" to upload a photo.}
print [</td></tr></table></center>]
print read %footer.html

; now display the member page with the new info refreshed:

wait :00:04
refresh-me: {
    <head><title></title>
    <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.html"></head>
}
print refresh-me
quit
```

Now we can finish up the rest of the work in step 3 of our outline. The pseudo code in my outline reads "The pages where the users edit their information will be forms that display the information currently in the selected entry, without the password. When the user submits the new password and updated info, the CGI checks that the submitted password matches the existing password for that entry, and then replaces the old block with the new one, in the database text file". I've already created links in index.html to reference the "edititemnumber" (created earlier using a counter variable in the foreach loop of index.cgi). And we've already created the basic data entry form to add new users. So we can check for the edititemnumber, and fill the form with appropriate items from the database. In order to find and replace the original entry in the database, once the user has made changes, the original values also need to be submitted as additional hidden form fields, along with the user-editable values in the form's text fields. Here's what I came up with:

```
if submitted/2 = "edititemnumber" [
    ; pick the correct entry from the database, using the submitted
    ; counter variable from the "edit" link in index.cgi:
    selected-block: pick bbs (
        (length? bbs) - (to-integer submitted/4) + 1
    )
    print [<font size=5>" Edit Your Existing Member Information:"</font>]
    print "          "
    ; here's a link we'll need for the section of the outline that
    ; enables image uploading:
    print rejoin [
        {<a href="./upload.cgi?name=} first selected-block
        {">Upload Image (Add or Change)</a><hr>}
    ]
    print "          "
    print "<br><br>"
    print {<strong><i>PASSWORD REQUIRED TO EDIT! </i></strong>
        (Enter it in the field below.)}
    print "<br><br>"
```

```
print [<FORM method="post" ACTION="./edit.cgi">]
print rejoin [
    {<br> Your Name:  <br>
        <input type=text size="60" name="username" value="}
     first selected-block {"><BR>}
]
print [<br> <strong> " Member Password " </strong> "(same
    as when you created the original account): " <br>
    <input type=text size="60"     name="password"><BR>
]
print rejoin [
    {<br> Address:  <br><input type=text size="60"
        name="address" value="}
    pick selected-block 3 {"><BR>}
]
print rejoin [
    {<br> Phone:  <br><input type=text size="60"
        name="phone" value="}
    pick selected-block 4 {"><BR>}
]
print rejoin [
    {<br> Email:  <br><input type=text size="60"
        name="email" value="}
    pick selected-block 5 {"><BR>}
]
print rejoin [
    {<br> Age:  <br><input type=text size="60"
        name="age" value="}
    pick selected-block 6 {"><BR>}
]
print rejoin [
    {<br> Language:  <br><input type=text size="60"
        name="language" value="}
    pick selected-block 7 {"><BR>}
]
print rejoin [
    {<br> Height:  <br><input type=text size="60"
        name="height" value="}
    pick selected-block 8 {"><BR><BR>}
]
print [" Other Information/Notes: " <br>]
print [<textarea name=otherinfo rows=5 cols=50>]
print [pick selected-block 9]
print [</textarea><BR><BR>]
print rejoin [
    {<input type="hidden" name="original_username" value="}
     pick selected-block 1 {">}
]
print rejoin [
    {<input type="hidden" name="original_date" value="}
    pick selected-block 2 {">}
]
print rejoin [
    {<input type="hidden" name="original_address" value="}
     pick selected-block 3 {">}
]
print rejoin [
    {<input type="hidden" name="original_phone" value="}
    pick selected-block 4 {">}
]
print rejoin [
    {<input type="hidden" name="original_email" value="}
    pick selected-block 5 {">}
```

```
    ]
    print rejoin [
        {<input type="hidden" name="original_age" value="}
        pick selected-block 6 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_language" value="}
        pick selected-block 7 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_height" value="}
        pick selected-block 8 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_otherinfo" value="}
        pick selected-block 9 {">}
    ]
    print [<INPUT TYPE="SUBMIT" NAME="Submit"
        VALUE="Update Member Information">]
    print [</FORM>]
    print [</td></tr></table></center>]
    print read %footer.html
    quit
]
```

I added the above code to index.cgi. Notice that the above form points to edit.cgi, which actually does the work of checking the password and processing the changes in the database. It has all the standard header and read-cgi code, and then it uses a foreach loop to look for a database entry that has all the same data as that submitted by the hidden items in the form above, and checks the original password in that entry. In comparing the original password with that entered by the user, I also enabled an administrator password "blahblah". I also added the code to email users their password, in case they've forgotten it (just send the stored password to the email address contained in the database, for that entry):

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print read %header.html

bbs: load %bb.db

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]
submitted: construct decode-cgi read-cgi

; get password from the entry submitted:

foreach message bbs [
    if all [
```

```
        find message submitted/original_username
        find message submitted/original_date
        find message submitted/original_address
        find message submitted/original_phone
        find message submitted/original_email
        find message submitted/original_age
        find message submitted/original_language
        find message submitted/original_height
        find message submitted/original_otherinfo
    ] [read-pass: message/10]
]

; save the old block:

old-message:  to-block reduce [
    submitted/original_username
    submitted/original_date
    submitted/original_address
    submitted/original_phone
    submitted/original_email
    submitted/original_age
    submitted/original_language
    submitted/original_height
    submitted/original_otherinfo
    read-pass
]

; so that the original pass is not replaced by "blahblah":

either submitted/password = "blahblah" [
    entered-pass: read-pass
] [
    entered-pass: submitted/password
]

; create the new entry for the database:

new-message:  to-block reduce [
    submitted/username
    submitted/original_date
    submitted/address
    submitted/phone
    submitted/email
    submitted/age
    submitted/language
    submitted/height
    submitted/otherinfo
    entered-pass
]

; check the password, and replace:

if submitted/password <> "" [
    either (
        read-pass = submitted/password
    ) or (
        submitted/password = "blahblah"
    ) [
        foreach message bbs [replace message old-message new-message]
    ] [
        print {
            <strong>Forgot your member password?</strong> <br><br>
            It's being emailed to the address for this entry, right now...
```

```
            Wait for this page to refresh, then <strong>check your email!
            </strong>
        }
        print read %footer.html
        wait 3
        set-net [user@website.com smtp.website.com]
        send (to-email submitted/original_email) (to-string rejoin [
            "Forgot your member password?" newline newline
            trim {Someone was editing an entry with this email address,
                but the incorrect password was used.  Here is the correct
                password, in case you've forgotten:}
             newline newline read-pass
        ])
    ]
]
save %bb.db bbs

; diplay the edited results on the main user page:

refresh-me: {
    <head><title></title>
    <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.cgi"></head>
}
print refresh-me
print read %footer.html
```

Here, I decided to add the backup code. What I did was create a folder for all previous versions of the database text file to be saved as backups. Then I created a text file that contained the number of the current backup file (to start out, that text file just contained the number 1). Then, I incremented that number and saved it back to that number file. And finally, I saved a copy of the current database to a text file with the current backup number appended to the filename. This code went right before bb.db was saved in the CGI above:

```
backup-num: load %backup-num.txt
backup-num: backup-num + 1
write %backup-num.txt backup-num
filename: to-file rejoin ["./backup/bb-" (to-string backup-num) ".txt"]
save filename bbs
```

The following code is basically a simpler version of the editing code above, which allows users to delete an entry. All that's needed in this case is the username and password. All the other info is passed along to delete.cgi as hidden fields. This code gets added to index.cgi:

```
if submitted/2 = "deleteitemnumber" [
    selected-block: pick bbs (
        (length? bbs) - (to-integer submitted/4) + 1
    )
    print [<font size=5>" Delete An Existing Member Account:"</font><hr>]
    print [<FORM method="post" ACTION="./delete.cgi">]
    print rejoin [
        {<br> Your Name:  <br>
            <input type=text size="60" name="username" value="}
        first selected-block {"><BR>}
    ]
    print [<br>" Member Password (
        same as when you created the original account): "
        <br><input type=text size="60"     name="password"><BR><BR>
    ]
    print rejoin [
```

```
        {<input type="hidden" name="original_username" value="}
        pick selected-block 1 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_date" value="}
        pick selected-block 2 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_address" value="}
        pick selected-block 3 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_phone" value="}
        pick selected-block 4 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_email" value="}
        pick selected-block 5 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_age" value="}
        pick selected-block 6 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_language" value="}
        pick selected-block 7 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_height" value="}
        pick selected-block 8 {">}
    ]
    print rejoin [
        {<input type="hidden" name="original_otherinfo" value="}
        pick selected-block 9 {">}
    ]
    print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Delete Member Info">]
    print [</FORM>]
    print [</td></tr></table></center>]
    print read %footer.html
    quit
]
```

Here's the code for delete.cgi, which the above form points to, and which does the actual work of deleting the selected block from the database (it's basically a variation of the edit.cgi script above):

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print read %header.html

bbs: load %bb.db

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
```

```
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]
submitted: construct decode-cgi read-cgi

foreach message bbs [
    if all [
        find message submitted/original_username
        find message submitted/original_date
        find message submitted/original_address
        find message submitted/original_phone
        find message submitted/original_email
        find message submitted/original_age
        find message submitted/original_language
        find message submitted/original_height
        find message submitted/original_otherinfo
    ] [read-pass: message/10]
]

old-message:  to-block reduce [
    submitted/original_username
    submitted/original_date
    submitted/original_address
    submitted/original_phone
    submitted/original_email
    submitted/original_age
    submitted/original_language
    submitted/original_height
    submitted/original_otherinfo
    read-pass
]

if submitted/password <> "" [
    if (
        read-pass = submitted/password
    ) or (
        submitted/password = "blahblah"
    ) [
        backup-num: load %backup-num.txt
        backup-num: backup-num + 1
        write %backup-num.txt backup-num
        filename: to-file rejoin [
            "./backup/bb-" (to-string backup-num) ".txt"
        ]
        save filename bbs

        foreach message bbs [replace message old-message ""]
    ]
]

remove-each message bbs [
    any [
        message = [""]
        (all [
            message/1 = "" message/2 = "" message/3 = "" message/4 = ""
            message/5 = "" message/6 = "" message/7 = "" message/8 = ""
            message/9 = ""
            ]
        )
    ]
]
```

```
save %bb.db bbs

refresh-me: {
    <head><title></title>
    <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.cgi"></head>
}
print refresh-me
print read %footer.html
```

Creating the image upload page for step #4 in our outline was a bit of a challenge. That's because REBOL has no built-in way to accept binary data from HTML forms (images, in this case), called "form-multipart" data. I searched the mailing list and quickly found a solution at http://www.rebol.org/cgi-bin/cgiwrap/rebol/ml-display-thread.r?m=rmlKVSQ. Andreas Bolka's "decode-multipart-form-data" did exactly what I needed. That function converts the data entered by a user, as well as the files they choose and upload from their hard drive, into a friendly and easy to use REBOL object.

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [Title: "HTTP File Upload"]
print "content-type: text/html^/"
print read %header.html

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

; here's Andreas's magic function to read form/multipart data:

decode-multipart-form-data: func [
    p-content-type
    p-post-data
    /local list ct bd delim-beg delim-end non-cr non-lf non-crlf mime-part
] [
    list: copy []
    if not found? find p-content-type "multipart/form-data" [return list]

    ct: copy p-content-type
    bd: join "--" copy find/tail ct "boundary="
    delim-beg: join bd crlf
    delim-end: join crlf bd

    non-cr:     complement charset reduce [ cr ]
    non-lf:     complement charset reduce [ newline ]
    non-crlf:   [ non-cr | cr non-lf ]
    mime-part:  [
        ( ct-dispo: content: none ct-type: "text/plain" )
        delim-beg ; mime-part start delimiter
        "content-disposition: " copy ct-dispo any non-crlf crlf
        opt [ "content-type: " copy ct-type any non-crlf crlf ]
        crlf ; content delimiter
```

```
        copy content
        to delim-end crlf ; mime-part end delimiter
        ( handle-mime-part ct-dispo ct-type content )
    ]

    handle-mime-part: func [
        p-ct-dispo
        p-ct-type
        p-content
        /local tmp name value val-p
    ] [
        p-ct-dispo: parse p-ct-dispo {;="}

        name: to-set-word (select p-ct-dispo "name")
        either (none? tmp: select p-ct-dispo "filename")
               and (found? find p-ct-type "text/plain") [
            value: content
        ] [
            value: make object! [
                filename: copy tmp
                type: copy p-ct-type
                content: either none? p-content [none] [copy p-content]
            ]
        ]

        either val-p: find list name
            [change/only next val-p compose [(first next val-p) (value)]]
            [ append list compose [ (to-set-word name) (value) ] ]
    ]

    use [ ct-dispo ct-type content ] [
        parse/all p-post-data [ some mime-part "--" crlf ]
    ]

    list
]

; now we can put the uploaded binary, and all the text entered by the
; user via the HTML form, into a REBOL object.  we can refer to the
; uploaded photo using the syntax:  cgi-object/photo/content

post-data: read-cgi
cgi-object: construct decode-multipart-form-data (
    system/options/cgi/content-type copy post-data
)

; I created a "./files" subdirectory to hold these images.  Now
; write the file to the web server using the original filename,
; but without any Windows path characters, and notify the user:

adjusted-filename: copy cgi-object/photo/filename
adjusted-filename: replace/all adjusted-filename "/" "-"
adjusted-filename: replace/all adjusted-filename "\" "-"
adjusted-filename: replace/all adjusted-filename " " "_"
adjusted-filename: replace/all adjusted-filename ":" "_"
adjusted-filename: to-file rejoin ["./files/" adjusted-filename]
write/binary adjusted-filename cgi-object/photo/content
print [<strong>]
print {Upload Complete.  }
print [</strong>]
print [<br><br>]

; now add an HTML link to this file, to the database:
```

```
bbs: load %bb.db
entry: copy []
link-added: rejoin [
    {<a href = "} to-string adjusted-filename {" target=_blank>}
    {<IMG align=baseline alt="" border=0 hspace=0 src="}
    to-string adjusted-filename
    {" width="160" height="120">} </a>
]  ; display image inline
append entry link-added
foreach message bbs [
    if (all [
        cgi-object/username = message/1
        cgi-object/password = message/10
    ]) [
        if ((length? message) < 11) [append message ""]
        message/11: entry
    ]
]
save %bb.db bbs

; show additions by refreshing the index.cgi page:

refresh-me: {
    <head><title></title>
    <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./index.cgi"></head>
}
print refresh-me
print read %footer.html
```

The last step in the outline was easy. I just used the code from the previous case study (the password protected CGI text editor), and pointed it to the database text file. I also looped through the backup directory and printed links to each of the files in that directory, so that any of the previous backup files could be easily copied and pasted into the editor, to revert the database to a previous state.

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Edit Database!!!"</TITLE></HEAD><BODY>]

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: decode-cgi read-cgi

; if schedule.txt has been edited and submitted:

if ((submitted/2 = "save") or (submitted/2 = "save")) [
```

```
    ; save newly edited schedule:
    write %./bb.db submitted/4
    print ["Database Saved."]
    ; print {<META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./bb.db">}
    quit
]

; if user is just opening page (i.e., no data has been submitted
; yet), request user/pass:

if ((submitted/2 = none) or (submitted/4 = none)) [
    print [<strong>"W A R N I N G  -  Private Server, Login Required:"
        </strong><BR><BR>]
    print [<FORM ACTION="./editor.cgi">]
    print [" Username: " <input type=text size="50" name="name"><BR><BR>]
    print [" Password: " <input type=text size="50" name="pass"><BR><BR>]
    print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
    print [</FORM>]
    quit
]

; check user/pass, end if incorrect:

response: false
if ((submitted/2 = "username") and (submitted/4 = "password")) [
    response: true
]
if response = false [print "Incorrect Username/Password." quit]

; if user/pass is ok, go on (backup before changes are made):

cur-time: to-string replace/all to-string now/time ":" "-"
schedule_text: read %./bb.db
write to-file rejoin [
    "./backup/" now/date "_" cur-time ".txt"
] schedule_text

; here's the form that lets the user edit the text:

print [<center>]
print [<strong>"Be sure to click [SUBMIT] when done:"</strong><BR><BR>]
print [<strong>"(This will OVERWRIGHT the current database!)"</strong>
    <BR><BR>]
print [<FORM method="post" ACTION="./editor.cgi">]
print [<INPUT TYPE=hidden NAME=submit_confirm VALUE="save">]
print [<textarea cols="100" rows="25" name="contents">]
print [schedule_text]
print [</textarea><BR><BR>]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</center>]
print {<br><br><br><br><br><br><br><br><hr>}

; here's a linked listing of all the backup files available for
; copy/paste:

foreach file (read %./backup/) [
    print rejoin [
        {<a href="./backup/} file {" target=_blank>} file {</a> }]
    ]
print [</BODY></HTML>]
```

That's it - the web site and all its features are complete! You can see a live demo at http://guitarz.org

/tester and download the complete set of scripts in this case study at http://guitarz.org/tester/member_board.zip.

10.12 Case 12 - A CGI Event Calendar

My friend liked the system above so much that we adapted it for use as an online classifieds page and also as an event calendar listing on the same web site. For the calendar, we just changed the database fields to: Event, Date/Time, Location, Contact Name, Contact Phone, Contact Email, Requirements. Links and display text such as "Join Now" were simply changed to "Enter A New Event", etc.

The calendar was in use for quite a while and functioning beautifully, when my friend asked if I could create an event page that actually looked like a normal calendar display, instead of just a list of events. Ok, so here's how I broke down the basic creative process:

1. Design an HTML page that looks like a calendar. My guiding thought was that the CGI program which printed this page would include a loop that runs through the days of the current month, and prints HTML table rows and cells for each numbered day, one row per group of days Sunday-Saturday.
2. For each day of the month printed in the table above, search through the database for dates that match the current table cell being printed, and then print the event description (first item in the block for that event), with a link to the event listing page.

As always, I began the process by looking for some existing code that may be useful in my design (it's always a good idea to avoid reinventing the wheel). My work was immediately made easy, when I searched for "calendar" at rebol.org. I quickly found the HTML calendar by Bohdan Lechnowsky, which prints out an HTML calendar display for the current month. It uses a table design created by loops, much like I had imagined. I read through Bohdan's code, made some comments as to what each section accomplished, and made some changes to the design of the tables so that the calendar stretched to fit the entire page of the browser. I also wrote a line of code to visually highlight the current day (so that today's date is always printed in a unique color). You can see the original code at the link above, and here are my tweaks and comments:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print {<HTML><HEAD><TITLE>Event Calendar</TITLE></HEAD><BODY>}

; print month + year header:

date: now/date
html: copy rejoin [
    {<CENTER><TABLE border=1 valign=middle width=99% height=99%>
        <TR><TD colspan=7 align=center height=8%><FONT size=5>}
    pick system/locale/months date/month {  } date/year
    {</FONT></TD></TR><TR>}
]

; print days header:

days: ["Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"]
foreach day days [
    append html rejoin [
        {<TD bgcolor="#206080" align=center width=10% height=5%>
        <FONT face="courier new,courier" color="FFFFFF" size="+1">}
        day
        {</FONT></TD>}
    ]
]
append html {</TR><TR>}

; print non-month days at the begining of month in gray:
```

```
sdate: date  sdate/day: 0
loop sdate/weekday // 7 + 1 [append html {<TD bgcolor=gray></TD>}]

; print every other day, with the current day in a unique color:

while [sdate/day: sdate/day + 1 sdate/month = date/month][
    append html rejoin [
        {<TD bgcolor="#}
        ; I ADDED THIS CODE TO VISUALLY HIGHLIGHT THE CURRENT DAY:
        either date/day = sdate/day ["AA9060"]["FFFFFF"]
        {" height=14% valign=top>} sdate/day
        {</TD>}
    ]
    if sdate/weekday = 6 [append HTML {</TR><TR>}]
]

; print non-month days at the end of month in gray:

loop 7 - sdate/weekday [append html rejoin [{<TD bgcolor=gray></TD>}]]

; finish and print:

append html {</TR></TABLE></CENTER></BODY></HTML>}
print html
```

With step 1 in my outline done, I completed the second and last step by adding the code below. It was really simple. First, I created a variable called "event-labels" which would hold any events in the database that occurred on a given day. I put this inside Bohdan's while loop, which ran through each day of the month and printed the calendar table cells for each separate day). I used a foreach loop to compare each date found in the database to the current date being added to the calendar. If there's a match, "event-labels" is rejoined with the first item in the event entry (the description of the event), and linked to the event display. The string of text in event-labels is then later printed into the table, within the current day's cell.

```
while [sdate/day: sdate/day + 1 sdate/month = date/month][
    event-labels: {}
    foreach entry bbs [
        date-in-entry: 1-Jan-1001
        attempt [date-in-entry: (to-date entry/3)]
        if (date-in-entry = sdate) [
            event-labels: rejoin [
                {<font size=1>}
                event-labels
                "<strong><br><br>"
                {<a href="http://website.com/path/calendar">}
                entry/1
                {</a>}
                "</strong>"
                {</font>}
            ]
        ]
    ]
```

That's it! Here's the whole script:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print {<HTML><HEAD><TITLE>Event Calendar</TITLE></HEAD><BODY>}
```

```
bbs: load %bb.db
date: now/date
html: copy rejoin [
    {<CENTER><TABLE border=1 valign=middle width=99% height=99%>
        <TR><TD colspan=7 align=center height=8%><FONT size=5>}
    pick system/locale/months date/month {  } date/year
    {</FONT></TD></TR><TR>}
]

days: ["Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"]
foreach day days [
    append html rejoin [
        {<TD bgcolor="#206080" align=center width=10% height=5%>
        <FONT face="courier new,courier" color="FFFFFF" size="+1">}
        day
        {</FONT></TD>}
    ]
]
append html {</TR><TR>}

sdate: date  sdate/day: 0
loop sdate/weekday // 7 + 1 [append html {<TD bgcolor=gray></TD>}]

while [sdate/day: sdate/day + 1 sdate/month = date/month][
    event-labels: {}
    foreach entry bbs [
        date-in-entry: 1-Jan-1001
        attempt [date-in-entry: (to-date entry/3)]
        if (date-in-entry = sdate) [
            event-labels: rejoin [
                {<font size=1>}
                event-labels
                "<strong><br><br>"
                {<a href="http://website.com/path/calendar">}
                entry/1
                {</a>}
                "</strong>"
                {</font>}
            ]
        ]
    ]
    append html rejoin [
        {<TD bgcolor="#}
        either date/day = sdate/day ["AA9060"]["FFFFFF"]
        ; HERE, THE EVENTS ARE PRINTED IN THE APPROPRIATE DAY:
        {" height=14% valign=top>} sdate/day event-labels
        {</TD>}
    ]
    if sdate/weekday = 6 [append html {</TR><TR>}]
]

loop 7 - sdate/weekday [append html rejoin [{<TD bgcolor=gray></TD>}]]

append html {</TR></TABLE></CENTER></BODY></HTML>}
print html
```

10.13 Case 13 - Ski Game, Snake Game, and Space Invaders Shootup

The Textris project was entertaining and educational, so I'm motivated to create another simple game. For this case study, I wanted to create a game that demonstrated more graphic techniques, instead of using text. I found a nice game tutorial at http://gm2d.com/2009/02/simple-flash-game-in-haxe. That game was written in another programming language, but does provide a nice example to emulate in REBOL.

In the Ski Game, the player is represented by a graphic that can be moved side to side across the top of the screen. Randomly placed tree images scroll up the screen, into the path of the skier. The goal is to avoid hitting trees for as long as possible. The longer the skier stays alive, the higher the score.

I began thinking through a plan of attack with this outline:

1. First, I'll need some images to use in the game. The tutorial link above contains open source graphics. I'll use the binary resouce embedder provided earlier in this tutorial to import and use those graphics in the code of this game.
2. I'll need to build a screen full of scrolling trees. To do that, here's some pseudo code, and a description of my imagined code outline: I'll create a graphical playing area using a 'draw' block on a 'view layout' window. To create a set of trees at various locations, I'll use a 'for' loop and the 'random' function to build up a block of the necessary enumerated coordinate positions, image data, etc. To move the trees, I'll use 'feel engage' on the draw block. Whenever a given amount of time has passed (some milliseconds, tested with a 'time' action in the engage loop), I'll increment the coordinate position of every tree, and increment the score. The majority of the program will run in this timer loop, so that the trees continuously move up the screen. If any of the trees reach the top of the screen, I'll remove them from the draw block, and replace them with new ones at random horizontal positions at the bottom of the screen. That will give the appearance of an endless hill of scrolling trees, and a continually counted score.
3. I'll need a player graphic. Side to side movement of that graphic needs to be controlled by the player, either via key strokes or mouse movements. I can either check for 'key actions in the engage loop above, or continuously check mouse position using the 'all-over' view feel option. If a movement left is detected, display a left facing skier graphic and update his position 1 pixel to the left (current-position - 1x0). If a movement right is detected, display a right facing graphic and update his right position 1 pixel to the right.
4. I'll need to check for collisions and end the game if the skier hits a tree. That'll involve comparing the positions of the skier graphic with those of *all* the tree graphics, in every iteration of the timer loop above.

For step one, I used Windows Paint to modify the right and left facing skier images that I found at the web site above. I created my own tree graphic by editing a simple line drawing found with Google. Using the binary resouce embedder from earlier in this tutorial, I converted those images to the following REBOL code:

```
tree:  load to-binary decompress 64#{
eJzt18sNwjAQBFDTBSVw5EQBnLjQE1XRngmBQEj8Wa/3M4oYOZKBKHkaWwTO1/sh
jDkNx3N6HI7LcOzCfnz/9v5cMnEai7lj4mokT9C7XczUsrhvGSku6RkgDIbHAEP0
2EiIMBdMDuaOWZCSL91bQvCsSY4MHE9umXz7ydVi3xgltYvEKboexzVSlpTa614d
NonpUauIv176dX0ZTRgJlVgzNl25A3gkGwld1bkrNFqqedQfEI02AU9PjDeMpac/
ShKeTXylROqCImlXRFd9zkQoh4tp+GpqlSTnLnum4HTEzK/gjpmTpDxSASlHFqYU
EE/8nddG9n+9LIm8t9OeIEra2JZWDRSG4VEioa0UFCZFqv/aMQh2Rf790EnGgcJU
SVAer0Bhcp7/epVJvkHzBHjPfz+XSe6BwryC5gmQno3mAY3tpba2KAAA
}
skier-left: load to-binary decompress 64#{
eJyN0U8og2EcB/DvNrz+E5fJZSmRf9Ej76h3Ne1AIspyMQflpJDFU/KO1cQmSnGa
A3PYkvInB3kvuyzlgJolh+fCRUq5iBvP8+5lTvKrX33ep+/zp9/b2Tthhl6zvGt5
W3nX8TYhS1//MOGnSjNEa/AUxd0UVQ3raL9IYbBvA2OBI9Q0DqB6fAujl08Yi97D
Hr3F5EQYSss2OrrWEFo5xB+VO5Vx/skvnxmQbDCFvxcjMJ/b0s6LAZXGA3O0ZtTt
pW3WbJmDeMC8a1gE9o3bTBFI9YvGhrOKSueyEQpu9ri60vQFXFqPMx1K+sNWrdOh
73Y/uMr85fKdcIrJ0z6vxSfsYV5KCU2JEPNIlD9dFZ65AfXwD+HsKdAZiiLdqtvt
```

```
Hh65E5ZklTGmDvWLgxxKkjAivwt7XxhJEvIsrCY8ikLs0Tj3yGeCKaQtdsX9fv3G
N1jCJdyv84lHJkNriiM7Li29OIDV0jcU8kuIHaiPLEDEsG9DQYxiQTi0A8sBpEvh
OT65GmBYH9Jx5nf8TFFUFf5ZX2hFdG1uAgAA
}
skier-right: load to-binary decompress 64#{
eJxz8s1jYgCDMiDWAGIJINYCYkYGFrD4D0YGOBBAMBn4++Yz6HjVMSgY1oP5gWdu
M/gHTmCwNutlKJ26l6F03VUGp3XnGGo+/mGILVnMoFkwhaHm7GcGz4m7GbABFwST
eQWSNXMQbM+3DAwlULbmEgaWXih75QUGzvkQJstMBwbPRRA2L1D5yS8QNudioNQF
qNYPDExAZRCtDg78c6Fa7wZK3Ycq940O3L1fAcLWigpctUsZzHTSj5Jd+l7NAKS6
3HnXk6jHSiBF7sUmxi7Gl9VAZrqVOxsZuTirg8TTS0qAQs5FIPF0BhYXFkgog/zg
7gJlq5SXpaWVF4O9lZKuXl6eVl4AZLIfKS82LzYuB2nlOFxWXl5ubA6ytm1KWU65
cXExkMl09lNNR3q5eTFQPYfHE7YT6cXlJgcYGI7cPMAOMtKhgcH9wE8FBuPycgOG
BoYKtl8ODL4gjccY2HSAfr4BVMvgAwyazwwsXSA7ORgY2BQYeH+Cw+sAKPo5wEHj
kQAO/GZwIIHDgc0AaxQSBAAFOXD7bgIAAA==
}
```

For all the rest of the steps in my initial outline, I organized my pseudo code thoughts into a general code outline. Since this program is all about a visual interface and event handling, I could use a very basic graphic and event handling code structure to begin filling out. Here's a simple skeleton:

```
; (Include the above graphic code here)

; Define some variables to start with (i.e., initial score = 0, etc).
; Create a "board" block to hold image information for all graphics
; to be display on screen.  Skier image should be first, then the trees.
; Use a "for" loop to create the data:

for i 1 20 1 [
   ; Add tree image data to the block described above.  Use the "random"
   ; function to come up with 20 random coordinate locations.
]

; Here's a basic screen layout structure with draw block, timer and
; key action detection, and the outline ideas above written into the
; appropriate areas of code:

view layout [
    scrn: box effect [draw board] rate 0 feel [
        engage: func [f a e] [
            if a = 'key [
                ; Move skier graphic left-right
            ]
            if a = 'time [
                ; Scroll the tree graphics upward.
                ; Remove any trees that go past the top of the screen.
                ; Replace removed trees with new trees at the bottom
                ; of the screen.
                ; Check for collisions and end the game if skier hits
                ; a tree.
                ; Update the score.
            ]
        ]
    ]
    ; Display a score in the GUI using some sort of text widget
]
```

Next, I fleshed out the code structure above with more detailed thoughts about how to accomplish everything in the initial descriptive outline. No actual code yet - just thoughts about how to accomplish each of the outline ideas, in the appropriate areas of the code structure. Here are my thoughts:

```
; (Include the above graphic code here)

; Define some variables to start with:

; I need to generate some random position coordinates.  Prepare (seed)
; the 'random function.

; All the items on the screen will be kept in a block (I'll call it
; "board").  Start with the code needed to display the skier image
; in a draw block.  The block should contain the following info for
; each image:

[
    the draw function 'image',
    the coordinate position of the graphic,
    the actual binary image data,
    the transparency color (black), so the edges of the images
        don't appear square (i.e., so the black outer frame corners of
        the image file disappear by blending into the background).
]

; Now add twenty trees to the above block, to appear at random places
; on the screen:

for i 1 20 1 [

    ; Assign a random coordinate to the variable 'pos', within the
    ; bounds of the playing screen.

    ; Shift every image position down 300 pixels, so the user
    ; has a moment to see them coming, and to get situated at the
    ; beginning of the game.

    ; Put each image into the "board" block described above.

]

; We now have a block of images that can be displayed on screen
; using 'draw' (see the '2D Drawing, Graphics, and Animation'
; section earlier in this tutorial).

; Center the GUI window, and get rid of the standard 20 pixel
; gray padding around the edges ('layout/tight'):

view center-face layout/tight [

    ; Set the color of the screen white like snow, and set the
    ; size of the playing area:

    scrn: box white 600x440 effect [draw board] rate 0 feel [
        engage: func [f a e] [
            if a = 'key [
                if e/key = 'right [

                    ; The second item in the block created above will
                    ; be the position coordinate of the skier graphic.
                    ; If the right arrow key has been pressed, add 5
                    ; to the horizontal portion of that position
                    ; coordinate.

                    ; The third item in the graphic block is the
                    ; actual graphic data used to display the skier.
```

```
                ; If the right arrow key has been pressed, that
                ; data should be replaced with the right facing
                ; skier graphic.

            ]
            if e/key = 'left [

                ; Same as the section above, but for the left key.

            ]

            ; Now that the data block has been updated with position
            ; and graphics alterations, show them on screen:

            show scrn
        ]
        if a = 'time [

            ; Everything in this block happens each time a timer
            ; action is detected in the feel block of the draw
            ; function above (currently, the rate is set to 0
            ; seconds, so all this code just keeps looping).

            ; First, move the trees up 5 pixels each.
            ; I'm going to need to deal with every item in the
            ; graphic block, sometimes removing and adding items.
            ; To make the whole process easier, I'm going to
            ; build a new copy of the changed block from scratch.

            ; I'll loop through each item in the existing block
            ; and check for the pair items (remember, there are 4
            ; items for each image ('image, coordinate pos, graphic
            ; data, and transparency color)).  Remember also that
            ; the first graphic in the block is the skier character.
            ; When working with tree graphics, we want to be sure to
            ; skip over the first four items in the block:

            foreach item board [

                ; Looping through the existing graphic block,
                ; subtract 0x5 from each tree's position (move each
                ; one up 5 pixels).  Append those new coordinate
                ; positions, along with every other item, in order,
                ; into the new block:

                either all [

                    ; If the item is a coordinate,
                    ; and we're not dealing with the first 4 items:

                ] [

                    ; Add the new coordinate position
                    ; (old position + 5) to the new block.

                ] [

                    ; Otherwise, add all other items to the new block,
                    ; as is (i.e., we're not changing the image or
                    ; transparency data).

                ]
```

```
        ; If the newly added coordinate is higher than the
        ; top of the screen, remove all 4 of its items from
        ; the new block (i.e., remove the tree graphic from
        ; the game).

    ]

    ; Now copy the new block back to the "board" variable.

    ; If any tree graphics have been removed from the top
    ; of the screen, replace them with new graphic
    ; data in the block.  We can check for removals by
    ; looking at the length of the data block.  It should
    ; be 84 items long (1 skier + 20 trees = 21*4, or 84
    ; items long).  Coordinates of the new trees should be
    ; at random horizontal locations along the bottom of
    ; the screen (i.e., somewhere along (random)x440 pixels).
    ; Append the new graphic data to the "board" block.

    ; Collision Detection:
    ;
    ; Check to see if skier position is within range of ANY
    ; tree position.  Use a foreach loop to make
    ; comparisons.  To ensure we're not detecting the
    ; skier colliding with himself, use a copy of the board
    ; without the first 4 items.  To check if images are
    ; touching, we need to consider the sizes and shapes of
    ; both the tree and skier graphics.  I came up with the
    ; following measurements:  If the top left corner of the
    ; skier image is within -40 and +15 horizontal pixels and
    ; 5 to 30 vertical pixel, they will be touching.
    ; This is an imperfect estimate that I came up with after
    ; some trial and error eyeballing the images, which
    ; considers the fact that we're starting calculations
    ; from the top left corner of each differently sized
    ; square image file.
    ; The calculation should check to see if horizontal
    ; skier_position - horizontal tree_position is within
    ; that range of pixels.

    ; I'll build a new block of data to perform the
    ; comparison, which has the skier items removed:

    collision-board: remove/part (copy board) 4
    foreach item collision-board [
        if (type? item) = pair! [
            if all [

              ; "item/1" and "item/2" refer respectively to
              ; horizontal and vertical components of the
              ; tree coordinate being checked (the x and y
              ; values in the coordinate).  "board/2/1" and
              ; "board/2/2" refer to the position of the
              ; skier graphic (remember, the skier's
              ; current position is always the second item
              ; in the block).  The calculations in this
              ; section will check the ranges described
              ; above.

            ] [
              ; If the above calculations evaluate to true,
              ; alert the user and end the game.
            ]
```

```
                    ]
                ]

                ; Every time through the loop, increase and update the
                ; score display.

            ]
        ]
    ]

    ; Put the text label "Score:" at the top left corner of the screen.

    ; The game score updated in the code above can just be contained in
    ; another text widget.  Assign that widget the word label "score".

    ; To make 'key actions work in the engage code above, the following
    ; stock line needs to be added to the layout:

    do [focus scrn]
]
```

Finally, I filled in the pseudo code outline above with actual working code that completed each pseudo code thought:

```
; (Include the above graphic code here)

; Define some variables to start with:

the-score: 0

; I need to generate some random position coordinates.  The following
; line is stock REBOL code to ensure that numbers are actually random:

random/seed now

; All the items on the screen will be kept in a block.
; Start with the code needed to display the skier image
; in a draw block (black is the transparent color):

board: reduce ['image 300x20 skier-right black]

; Now add twenty trees to the above block, to appear at random places
; on the screen:

for i 1 20 1 [

    ; Assign a random coordinate to the variable 'pos', within the
    ; bounds of the playing screen:

    pos: random 600x540

    ; Shift every image position down 300 pixels, so the user
    ; has a second to see them coming, and to get situated at the
    ; beginning of the game:

    pos: pos + 0x300

    ; Put each image into the block above:

    append board reduce ['image pos tree black]
]
```

```
; We now have a block of images that can be displayed on screen
; using 'draw' (see the '2D Drawing, Graphics, and Animation'
; section earlier in this tutorial).

; Center the GUI window, and get rid of the standard 20 pixel
; gray padding around the edges ('layout/tight'):

view center-face layout/tight [

    ; Set the color of the screen white like snow, and set the
    ; size of the playing area:

    scrn: box white 600x440 effect [draw board] rate 0 feel [
        engage: func [f a e] [
            if a = 'key [
                if e/key = 'right [

                    ; The second item in the block created above is
                    ; the position coordinate of the skier graphic.
                    ; If the right arrow key has been pressed, add 5
                    ; to the horizontal portion of that position
                    ; coordinate:

                    board/2: board/2 + 5x0

                    ; The second item in the graphic block is the
                    ; actual graphic data used to display the skier.
                    ; If the right arrow key has been pressed, that
                    ; data should be replaced with th right facing
                    ; skier graphic:

                    board/3: skier-right
                ]
                if e/key = 'left [

                    ; Same as the section above, but for the left key:

                    board/2: board/2 - 5x0
                    board/3: skier-left
                ]

                ; Now that the data block has been updated with position
                ; and graphics alterations, show them on screen:

                show scrn
            ]
            if a = 'time [

                ; Everything in this block happens each time a timer
                ; action is detected in the feel block of the draw
                ; function (currently, the rate is set to 0 seconds,
                ; so this code all just keeps looping).

                ; First, move the trees up 5 pixels each.
                ; I'm going to need to deal with every item in the
                ; graphic block, sometimes removing and adding items.
                ; To make the whole process easier, I'm going to
                ; build a new copy of the changed block from scratch:

                new-board: copy []

                ; Now I'll loop through each item in the existing block
```

```
; and check for the pair items (remember, there are 4
; items for each image ('image, coordinate pos, graphic
; data, and transparency color)).  Remember also that
; the first graphic in the block is the skier character.
; When working with tree graphics, we want to be sure to
; skip over the first four items in the block:

foreach item board [

    ; Looping through the existing graphic block,
    ; subtract 0x5 from each tree's position (move each
    ; one up 5 pixels).  Append those new coordinate
    ; positions, along with every item, in order, into
    ; the new block:

    either all [

        ; If the item is a coordinate:

        ((type? item) = pair!)

        ; and we're not dealing with the first 4 items:
        ; (after we're done with this loop, 4 items will
        ; have been added to the new block):

        ((length? new-board) > 4)
    ] [

        ; Add the moved up position to the new block:

        append new-board (item - 0x5)
    ] [

        ; Add all other items to the new block:

        append new-board item
    ]

    ; If the newly added coordinate is higher than the
    ; top of the screen, remove all 4 of its items from
    ; the new block (i.e., remove the tree graphic from
    ; the game):

    coord: first back back (tail new-board)
    if ((type? coord) = pair!) [
        if ((second coord) < -60) [
            remove back tail new-board
            remove back tail new-board
            remove back tail new-board
            remove back tail new-board
        ]
    ]
]

; Now copy the new block back to the "board" variable:

board: copy new-board

; If any tree graphics have been removed from the top
; of the screen, replace them in the with new graphic
; data in the block.  We can check for removals by
; looking at the length of the data block.  It should
; be 84 items long (1 skier + 20 trees = 21*4, or 84
```

```
; items long).  Coordinates of the new trees should be
; at random horizontal locations along the bottom of
; the screen (i.e., somewhere along (random)x440 pixels):
; Append the new graphic to the screen:

if (length? new-board) < 84 [
    column: random 600
    pos: to-pair rejoin [column "x" 440]
    append board reduce ['image pos tree black]
]

; Collision Detection:
;
; Check to see if skier position is within range of ANY
; tree position.  Use a foreach loop to make
; comparisons.  To make ensure you're not detecting the
; skier colliding with himself, use a copy of the board
; without the first 4 items.  To check if images are
; touching, we need to consider the sizes and shapes of
; both the tree and skier graphics.  I came up with the
; following measurements:  If the top left corner of the
; skier image is within -40 and +15 horizontal pixels and
; 5 to 30 vertical pixel, they will be touching.
; This is an imperfect estimate that I came up with some
; trial and error eyeballing of the images, and which
; considers the fact that we're starting the calculations
; from the top left corner of each differently sized
; image.
; The calculation should check to see if horizontal
; skier_position - horizontal tree_position is within
; that range of pixels.


collision-board: remove/part (copy board) 4
foreach item collision-board [
    if (type? item) = pair! [
        if all [

            ; "item/1" and "item/2" refer respectively to
            ; horizontal and vertical components of the
            ; tree coordinate being checked (the x and y
            ; values in the coordinate).  "board/2/1" and
            ; "board/2/2" refer to the position of the
            ; skier graphic (remember, the skier's
            ; current position is always the second item
            ; in the block).  The calculations below
            ; check the ranges described above:

            ((item/1 - board/2/1) < 15)
            ((item/1 - board/2/1) > -40)
            ((board/2/2 - item/2) < 30)
            ((board/2/2 - item/2) > 5)
        ] [

            ; Alert the user and end the game:

            alert "Ouch - you hit a tree!"
            alert rejoin ["Final Score: " the-score]
            quit
        ]
    ]
]
```

```
                ; Every time through the loop, increase and update the
                ; score display:

                the-score: the-score + 1
                score/text: to-string the-score
                show scrn
            ]
        ]
    ]

    ; Put the word "Score:" at the top left corner of the screen:

    origin across h2 "Score:"

    ; Here's the game score text which is updated in the code above,
    ; It's just a text header widget, assign the word label "score":

    score: h2 bold "000000"

    ; To make 'key actions work in the engage code above, the following
    ; stock line needs to be added to the layout:

    do [focus scrn]
]
```

Here's the final game, with comments removed:

```
REBOL [title: "Ski Game"]

tree:  load to-binary decompress 64#{
eJzt18sNwjAQBFDTBSVw5EQBnLjQE1XRngmBQEj8Wa/3M4oYOZKBKHkaWwTO1/sh
jDkNx3N6HI7LcOzCfnz/9v5cMnEai7lj4mokT9C7XczUsrhvGSku6RkgDIbHAEP0
2EiIMBdMDuaOWZCSL91bQvCsSY4MHE9umXz7ydVi3xgltYvEKboexzVSlpTa614d
NonpUauIv176dX0ZTRgJlVgzNl25A3gkGwld1bkrNFqqedQfEI02AU9PjDeMpac/
ShKeTXylROqCImlXRFd9zkQoh4tp+GpqlSTnLnum4HTEzK/gjpmTpDxSASlHFqYU
EE/8nddG9n+9LIm8t9OeIEra2JZWDRSG4VEioa0UFCZFqv/aMQh2Rf790EnGgcJU
SVAer0Bhcp7/epVJvkHzBHjPfz+XSe6BwryC5gmQno3mAY3tpba2KAAA
}
skier-left: load to-binary decompress 64#{
eJyN0U8og2EcB/DvNrz+E5fJZSmRf9Ej76h3Ne1AIspyMQflpJDFU/KO1cQmSnGa
A3PYkvInB3kvuyzlgJolh+fCRUq5iBvP8+5lTvKrX33ep+/zp9/b2Tthhl6zvGt5
W3nX8TYhS1//MOGnSjNEa/AUxd0UVQ3raL9IYbBvA2OBI9Q0DqB6fAujl08Yi97D
Hr3F5EQYSss2OrrWEFo5xB+VO5Vx/skvnxmQbDCFvxcjMJ/b0s6LAZXGA3O0ZtTt
pW3WbJmDeMC8a1gE9o3bTBFI9YvGhrOKSueyEQpu9ri60vQFXFqPMx1K+sNWrdOh
73Y/uMr85fKdcIrJ0z6vxSfsYV5KCU2JEPNIlD9dFZ65AfXwD+HsKdAZiiLdqtvt
Hh65E5ZklTGmDvWLgxxKkjAivwt7XxhJEvIsrCY8ikLs0Tj3yGeCKaQtdsX9fv3G
N1jCJdyv84lHJkNriiM7Li29OIDV0jcU8kuIHaiPLEDEsG9DQYxiQTi0A8sBpEvh
OT65GmBYH9Jx5nf8TFFUFf5ZX2hFdG1uAgAA
}
skier-right: load to-binary decompress 64#{
eJxz8s1jYgCDMiDWAGIJINYCYkYGFrD4D0YGOBBAMBn4++Yz6HjVMSgY1oP5gWdu
M/gHTmCwNutlKJ26l6F03VUGp3XnGGo+/mGILVnMoFkwhaHm7GcGz4m7GbABFwST
eQWSNXMQbM+3DAwlULbmEgaWXih75QUGzvkQJstMBwbPRRA2L1D5yS8QNudioNQF
qNYPDExAZRCtDg78c6Fa7wZK3Ycq940O3L1fAcLWigpctUsZzHTSj5Jd+l7NAKS6
3HnXk6jHSiBF7sUmxi7Gl9VAZrqVOxsZuTirg8TTS0qAQs5FIPF0BhYXFkgog/zg
7gJlq5SXpaWVF4O9lZKuXl6eVl4AZLIfKS82LzYuB2nlOFxWXl5ubA6ytm1KWU65
cXExkMl09lNNR3q5eTFQPYfHE7YT6cXlJgcYGI7cPMAOMtKhgcH9wE8FBuPycgOG
BoYKtl8ODL4gjccY2HSAfr4BVMvgAwyazwwsXSA7ORgY2BQYeH+Cw+sAKPo5wEHj
kQAO/GZwIIHDgc0AaxQSBAAFOXD7bgIAAA==
}
random/seed now
```

```
the-score: 0
board: reduce ['image 300x20 skier-right black]
for i 1 20 1 [
    pos: random 600x540
    pos: pos + 0x300
    append board reduce ['image pos tree black]
]
view center-face layout/tight [
    scrn: box white 600x440 effect [draw board] rate 0 feel [
        engage: func [f a e] [
            if a = 'key [
                if e/key = 'right [
                    board/2: board/2 + 5x0
                    board/3: skier-right
                ]
                if e/key = 'left [
                    board/2: board/2 - 5x0
                    board/3: skier-left
                ]
                show scrn
            ]
            if a = 'time [
                new-board: copy []
                foreach item board [
                    either all [
                        ((type? item) = pair!)
                        ((length? new-board) > 4)
                    ] [
                        append new-board (item - 0x5)
                    ] [
                        append new-board item
                    ]
                    coord: first back back (tail new-board)
                    if ((type? coord) = pair!) [
                        if ((second coord) < -60) [
                            remove back tail new-board
                            remove back tail new-board
                            remove back tail new-board
                            remove back tail new-board
                        ]
                    ]
                ]
                board: copy new-board
                if (length? new-board) < 84 [
                    column: random 600
                    pos: to-pair rejoin [column "x" 440]
                    append board reduce ['image pos tree black]
                ]
                collision-board: remove/part (copy board) 4
                foreach item collision-board [
                    if (type? item) = pair! [
                        if all [
                          ((item/1 - board/2/1) < 15)
                          ((item/1 - board/2/1) > -40)
                          ((board/2/2 - item/2) < 30)
                          ((board/2/2 - item/2) > 5)
                        ] [
                            alert "Ouch - you hit a tree!"
                            alert rejoin ["Final Score: " the-score]
                            quit
                        ]
                    ]
                ]
```

```
                the-score: the-score + 1
                score/text: to-string the-score
                show scrn
            ]
        ]
    ]
    origin across h2 "Score:"
    score: h2 bold "000000"
    do [focus scrn]
]
```

10.13.1 Addendum

It should be noted that I did lots of trial and error coding along the way, while writing and testing this program. One thing that I tried initially was to have the skier move left-right by following left-right mouse gestures. I scrapped that idea because my code performed too slowly for this application, but the resulting code may still be useful in other projects. It's included here for completeness.

I defined this starting variable at the beginning of the program:

```
mouse-pos: 0x0
```

and added this code to the "feel" block, directly beneath the "engage" function:

```
over: func [f a p] [
    if not mouse-pos = p [  ; i.e., if mouse has moved
        either p/1 > mouse-pos/1 [  ; true = mouse has moved right
            ; update the skier image data in the "board" block:
            board/3: skier-right
        ] [
            board/3: skier-left
        ]
        ; set skier's position based on the column position of the
        ; mouse:
        board/2: to-pair rejoin compose [(p/1 - 35) "x" 20]
        mouse-pos: p
        show scrn
    ]
]
```

In order for the REBOL to continuously check for mouse events, the following "all-over" option must be added to the 'view layout' code:

```
view/options layout [...] [all-over]
```

Remove the key action code in the engage function, and replace it with the above changes. The skier will move left-right based upon left-right movements of the mouse.

Another way to accomplish the same goal, without using the "all-over" option, is to use the feel "detect" function:

```
detect: func [f e] [
    if e/type = 'move [
        p: e/offset
        if not mouse-pos = p [
            either p/1 > mouse-pos/1 [
```

```
                board/3: skier-right
            ] [
                board/3: skier-left
            ]
            board/2: to-pair rejoin compose [(p/1 - 35) "x" 20]
            mouse-pos: p
            show scrn
        ]
    ]
    e
]
```

That type of mouse control wasn't the best solution here, but could certainly be useful in other programs.

10.13.2 Snake Game

Below is the code for the classic "Snake" game. The point of the snake game is to move a snake image around the screen, devouring a food pellet that appears at random locations. Every time you eat a pellet, your snake body grows by one unit. Avoid hitting the edge of the playing field, and avoid hitting your own body for as long as possible.

Notice that the code outline for this program is nearly identical to that of the ski game:

1. Embed the images needed to display snake sections and food pellets (for this example, I used simple green and red button images created using the "to-image" and "layout" functions, but that can be easily changed).
2. Set some initial variables (starting score, random starting coordinates, initial values for flags used throughout the program, etc.).
3. Create a board block to hold the image data and coordinates of the snake sections and food images.
4. Display the playing board in a "view layout" draw block, and move game play along by continuously checking for "feel engage" time and key events.
5. Change the direction the snake moves every time a key is pressed. Adjust snake coordinates (move the snake section images), and adjust the score, every time a timer event occurs (15 times per second). As in the ski game, create a temporary block to copy and adjust all the new snake coordinates (move the head of the snake to a new adjacent location, then move each consecutive section of the snake to the previous location of its adjoining section).
6. Check for collisions by comparing coordinate positions of the snake sections with other items on the board. End the game if the snake collides with a wall, or itself. Whenever the snake collides with a food image, move the food image to a new random coordinate, and add a new snake section image to the board (append an image to the board block, to increase the length of the snake).

```
REBOL [Title: "Snake Game"]

snake: to-image layout/tight [button red 10x10]
food: to-image layout/tight [button green 10x10]
the-score: 0  direction: 0x10  newsection: false  random/seed now
rand-pair: func [s] [
    to-pair rejoin [(round/to random s 10) "x" (round/to random s 10)]
]
b: reduce [
    'image food ((rand-pair 190) + 50x50)
    'image snake ((rand-pair 190) + 50x50)
]
view center-face layout/tight gui: [
    scrn: box white 300x300 effect [draw b] rate 15 feel [
        engage: func [f a e] [
            if a = 'key [
```

```
                if e/key = 'up [direction: 0x-10]
                if e/key = 'down [direction: 0x10]
                if e/key = 'left [direction: -10x0]
                if e/key = 'right [direction: 10x0]
            ]
            if a = 'time [
                if any [b/6/1 < 0 b/6/2 < 0 b/6/1 > 290 b/6/2 > 290] [
                    alert "You hit the wall!" quit
                ]
                if find (at b 7) b/6 [alert "You hit yourself!" quit]
                if within? b/6 b/3 10x10 [
                    append b reduce ['image snake (last b)]
                    newsection: true
                    b/3: (rand-pair 290)
                ]
                newb: copy/part head b 5  append newb (b/6 + direction)
                for item 7 (length? head b) 1 [
                    either (type? (pick b item) = pair!) [
                        append newb pick b (item - 3)
                    ] [
                        append newb pick b item
                    ]
                ]
                if newsection = true [
                    clear (back tail newb)
                    append newb (last b)
                    newsection: false
                ]
                b: copy newb
                show scrn
                the-score: the-score + 1
                score/text: to-string the-score
            ]
        ]
    ]
    origin across h2 "Score:"
    score: h2 bold "000000"
    do [focus scrn]
]
```

10.13.3 Obfuscation

Just for fun, I created an obfuscated (unreadable) version of the snake program. REBOL is such a malleable language that it's possible to create unbelievably compact code. I was able to squash the above 2030 bytes of nicely formatted code into the following 771 bytes of pure REBOL fury:

```
do[p: :append u: :reduce k: :pick r: :random y: :layout q: 'image z: :if
g: :to-image v: :length? x: does[alert join{SCORE: }[v b]quit]s: g y/tight
[btn red 10x10]o: g y/tight[btn tan 10x10]d: 0x10 w: 0 r/seed now b: u[q
o(((r 19x19)* 10)+ 50x50)q s(((r 19x19)* 10)+ 50x50)]view center-face
y/tight[c: area 305x305 effect[draw b]rate 15 feel[engage: func[f a e][z a
= 'key[d: select u['up 0x-10 'down 0x10 'left -10x0 'right 10x0]e/key]z a
= 'time[z any[b/6/1 < 0 b/6/2 < 0 b/6/1 > 290 b/6/2 > 290][x]z find(at b
7)b/6[x]z within? b/6 b/3 10x10[p b u[q s(last b)]w: 1 b/3:((r 29x29)*
10)]n: copy/part b 5 p n(b/6 + d)for i 7(v b)1 [either(type?(k b i)=
pair!)[p n k b(i - 3)][p n k b i]]z w = 1[clear(back tail n)p n(last b)w:
0]b: copy n show c]]]do[focus c]]]
```

The above code is a fully functional snake program (go ahead, paste it into the interpreter...). I created it by renaming functions using single-letter labels (r: :random, p: :append, etc.). Any function that was used several times in the program got renamed. I also removed any spaces which

surrounded parentheses or brackets. Line breaks are included only so that the code fits inside this web page - otherwise, they're not necessary. There's not much practical purpose to obfuscating code in this way, but it can be used to impress all your friends who don't know REBOL :)

10.13.4 Space Invaders Shootup

Below is an extremely simple variation of the classic Space Invaders game idea. Compare the code outline of this program with that of the previous games, and notice again the similar structure: embedded image definitions, game board block creation, view layout draw display, "feel engage" key and time event loops, coodinate calculations to move game images and to detect collisions, etc. Notice also the "system/view/caret: none" and "system/view/caret: head f/text" code before and after each "show scrn". This erases the text caret (small vertical line) that appears in a face whenever the "focus" function is called:

```
REBOL [title: "Space Invaders Shootup"]

alien1:  load to-binary decompress 64#{
eJx9UzFLQzEQjijUOognHTIVhCd0cXJ1kLe3g7SbFKcsWQoWZ7MFhNKxg0PpH3Cx
WbKUqpPoUNcOPim1Q+kPkCJekvf0NTx7cLl7d8l33+XywvL+FrFyhVpCPUY9QN0g
LnG7ScjjrtM98iedToeM3kbW7/f71k4/p6R+USe9Xo/UqjUbi94jMhgMrL/8XpLm
ZZP4spPyzxVTT35MM2Zir4vFYu4dM7GP2M483Fa8f8w0O/Vy24yzo8RXipfJmdb8
kJxwrdJ7K4gxiSs7/09czYpdW6vcsI+AtrEKQ7ScDPlLHO/aNQ8huzaVeSDaHrNi
3IlBjDI6mqVsWvIA0E5ZJ2OtlUIuAKHmqoS5kHOt9UPMP0sm3TU5PHdHQVIZMs3v
qZTPmrMAQAj6ZXOSUtkwPKRwloKQNlexCDOvR4fpclGq76KNzC2mQPiG681i5gAw
ZusVJEAh5JojBzrEGQYC2dncuh7+y83d7ASVAu8MpAQqkT9+3Gg3Q+wHI2AZSAFm
1+99FzMQkzllVUxeTFUrc4vC4Q4VV4wlLyaerjD1XPe+tLxK8SNbqTrJOIf/Bd4X
V+VU7AfjSm0ZEgQAAA==
}
ship1: load to-binary decompress 64#{
eJx1Us9L3EAU/rTbMdHE9VlYukSQFhUpvXjQXrfizR8XkYAnt0oQVsTLGlgEFdSL
l3jqpXgre6sEJAgDsidPIul/sHjopeIfILj0JdnV3ez0y7zJzLx533vvY2YXhzOI
scs2yfaF7QNbDxLHjzfAu4HEhpKryAgDfccC4rfAws0IjF/96HsWyH7XMNXIon9Z
QJvWsNQw8XZP4NPlKD73Whi/HcZO4z207fv7jyo8/jk4r1TdFQXcSrV+flEtq3x2
5amuB44lyU+BpHRKHq4dKXnZCbLkxl9kOF5BarPVDFWyBAWcEAVFsjrhENGmhyPK
UXe+XNHf9HqZW9GgyzUUoloqXcXE1wv6iSTHohSkQ8yJQ5l2RCiSvPIGbaVkTFuu
Ge5/erfdurb+wM3ETZHPyjaX5NzNHPATOHMsn894sMZJWX4uH78OYSvTrUU+paI2
q8nQl5JHMFaSOZLBbHPnoR2ndHUa5NtPwubfFKziT1YqRDdY2VV3JckT3X2ZIlwW
KQjmUxGhGQ0Ecm5OlhBvUsSi/NpXmjLRoFx4YWuL0789fN24m+jsK2x+wGE+JjLR
DePiqdbKZqZojf1qLZ2ptdO3ZrxXwjCODzuThK3Af4EF8jYSBAAA
}
alien-fire: load to-binary decompress 64#{
eJxz8o1jZACDMiDWAGI+IJYFYkYGFrD4CyAW5oZgAYhSBhZmFoaWphaG48eOMwQF
BDFoaGgwPH36lGHZsmUM4uLiDFk5WQyzZs1iuHHzBsOfv38Ydu7cyWBhZsFQXlrO
EBEVATTBaWlolAoDA/vp3bt37wHyZwPpTUCaedqpUBWGS6HLMj8AedpA0Z1QGqTK
KXrNtCdgF/BLtrCD6GywOAPDabA6BobCTAMwXTfzFMh8uM7ZUBpi/p3QZdMMwLp2
796GbH7omrR2sH6Omc+h5m4C09pQuiKzHWp+O1R+D1QeQjstPQINIwag+wBUhlwj
XgEAAA==
}
ship-fire: load to-binary decompress 64#{
eJxz8t3FAAFlQKwBxOxALAjEjAwsYHEXIBbmhmABqFo2FhYG9l4eBvajbAwKSTIM
/H8FGFjUOBg4tnEyGP1VYWAXZWOwadNg4KhiYdA5JMLAacbJIHNLhUFnkgiDIpMg
2IyDd2UYVMqdGNLLyxoOz7RpCJ5p2pDi4sYAwlFpSz+AcEoJkF8O5KstZWhUkvig
4uLEoAIUO7f7zQcA8m8lvboAAAA=
}
bottom: 270  end: sidewall: false  random/seed now
b: ['image 300x400 ship1 'line -10x270 610x270]
for row 60 220 40 [
    for column 20 380 60 [
        pos: to-pair rejoin [column "x" row]
        append b reduce ['image pos alien1]
    ]
```

```
]
view center-face layout/tight [
    scrn: box black 600x440 effect [draw b] rate 1000 feel [
        engage: func [f a e] [
            if a = 'key [
                if e/key = 'right [b/2: b/2 + 5x0]
                if e/key = 'left [b/2: b/2 - 5x0]
                if e/key = 'up [
                    if not find b ship-fire [
                        fire-pos: b/2 + 25x-20
                        append b reduce ['image fire-pos ship-fire]
                    ]
                ]
                system/view/caret: none
                show scrn
                system/view/caret: head f/text
            ]
            if a = 'time [
                if (random 1000) > 900 [
                    f-pos: to-pair rejoin [random 600 "x" bottom]
                    append b reduce ['image f-pos alien-fire]
                ]
                for i 1 (length? b) 1 [
                    removed: false
                    if ((pick b i) = ship-fire) [
                        for c 8 (length? head b) 3 [
                            if (within? (pick b c) (
                            (pick b (i - 1)) + -40x0) 50x35)
                            and ((pick b (c + 1)) <> ship-fire) [
                                removed: true
                                d: c
                                e: i - 1
                            ]
                        ]
                        either ((second (pick b (i - 1))) < -10) [
                            remove/part at b (i - 2) 3
                        ] [
                            do compose [b/(i - 1): b/(i - 1) - 0x9]
                        ]
                    ]
                    if ((pick b i) = alien1) [
                        either ((second (pick b (i - 1))) > 385) [
                            end: true
                        ] [
                            if ((first (pick b (i - 1))) > 550) [
                                sidewall: true
                                for item 4 (length? b) 1 [
                                    if (pick b item) = alien1 [
                                        do compose [
                                          b/(item - 1): b/(item - 1) + 0x2
                                        ]
                                    ]
                                ]
                                bottom: bottom + 2
                                b/5: to-pair rejoin [-10 "x" bottom]
                                b/6: to-pair rejoin [610 "x" bottom]
                            ]
                            if ((first (pick b (i - 1))) < 0) [
                                sidewall: false
                                for item 4 (length? b) 1 [
                                    if (pick b item) = alien1 [
                                        do compose [
                                          b/(item - 1): b/(item - 1) + 0x2
```

```
                                        ]
                                    ]
                                ]
                                bottom: bottom + 2
                                b/5: to-pair rejoin [-10 "x" bottom]
                                b/6: to-pair rejoin [610 "x" bottom]
                            ]
                            if sidewall = true [
                                do compose [b/(i - 1): b/(i - 1) - 2x0]
                            ]
                            if sidewall = false [
                                do compose [b/(i - 1): b/(i - 1) + 2x0]
                            ]
                        ]
                    ]
                    if ((pick b i) = alien-fire) [
                        if within? ((pick b (i - 1)) + 0x14) (
                            (pick b 2) + -10x0) 65x35 [
                            alert "You've been killed by alien fire!" quit
                        ]
                        either ((second (pick b (i - 1))) > 400) [
                            remove/part at b (i - 2) 3
                        ] [
                            do compose [b/(i - 1): b/(i - 1) + 0x3]
                        ]
                    ]
                    if removed = true [
                        remove/part (at b (d - 1)) 3
                        remove/part (at b (e - 1)) 3
                    ]
                ]
                system/view/caret: none
                show scrn
                system/view/caret: head f/text
                if not (find b alien1) [
                    alert "You killed all the aliens. You win!" quit
                ]
                if end = true [alert "The aliens landed! Game over." quit]
            ]
        ]
    ]
    do [focus scrn]
]
```

I created a version of this game for my fiance using an image of my face for ship1, and an image of her face as alien1. An XpackerX executable version of it is available at http://musiclessonz.com/rebol_tutorial/corina_invaders.exe.

Now take a break from coding, and play a few games :)

10.14 Case 14 - Media Player (Wave/Mp3 Jukebox)

This case study started when a reader of this tutorial sent me an email question. He was having trouble creating a simple script that would load the file names from a directory on his hard drive into a GUI text list. He wanted to be able to click on .wav files in the list in order to play the sounds. I generally don't have time to answer questions like that, but this one was a quicky. The following code switches to the Windows media folder, reads the directory listing, and displays the file names in a GUI text list:

```
change-dir %/c/Windows/media
view layout [text-list data read %.]
```

I just added the contents of the "play-sound" function found earlier in this tutorial, to the action block of the text list. This loads the contents of the value selected in the text list (the file name), and plays the sound:

```
change-dir %/c/Windows/media
view layout [
    vh2 "Click a File to Play:"
    text-list data read %. [
        wait 0
        sound-port: open sound://
        insert sound-port (load value)
        wait sound-port
        close sound-port
    ]
]
```

That was simple.

A few days later the reader emailed me for some additional help. As it stands, the script crashes if the user selects anything other than a .wav file, or if another file is selected while a .wav is currently playing. I wrote back with some code to get the text list to show only .wav files and to make the program wait to play another file. I also wrote some additional code to let users select a different starting directory. Here it is:

```
; Here's how to use the "request-dir" function to let the user
; select a folder to switch into:

start-dir: request-dir/dir %/c/Windows/media
change-dir start-dir

; To display only files with a ".wav" extension in your text list,
; create a new empty block.  Use a "foreach" loop to go through the
; directory listing, and append to the new block only file names
; which have ".wav" as the suffix:

waves: []
foreach file read %. [
    if %.wav = (suffix? file) [append waves file]
]

; Now you can display the "waves" block of data in the GUI text list.

; To wait for sounds to finish playing before another file can
; be selected, add a "wait-flag" variable to the play-sound
; function.  When a sound starts playing, set the wait-flag
```

```
; variable to true.  When it's finished playing, set the wait-flag
; to false.  Also be sure to set it initially to "false" at
; the beginning of your program:

play-sound: func [sound-file] [
    wait 0
    wait-flag: true
    ring: load sound-file
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
    wait-flag: false
]
wait-flag: false

; When a file is selected from the text list, only run the
; "play-sound" function if the "wait-flag" variable is not
; set to true (i.e., if no sounds are playing):

view layout [
    vh2 "Click a File to Play:"
    text-list data waves [
        if wait-flag <> true [
            play-sound value
        ]
    ]
]
```

As I tested the above code, I realized that a few various .wav files in the Windows media folder wouldn't play properly, and the script crashed. I added the following code to handle errors:

```
if error? try [play-sound value] [
    alert "malformed wave"     ; Alert the user with a message,
    close sound-port           ; close the port opened by the broken
    wait-flag: false           ; play-sound function, and set the flag
]                              ; back to false (so other waves can play).
```

I also decided to add a button to the GUI to allow users to change the directory at will, instead of just at the beginning of the script:

```
btn "Change Folder" [
    change-dir request-dir
    waves: copy []
    foreach file read %. [
        if %.wav = suffix? file [append waves file]
    ]
    file-list/data: waves
    show file-list
]
```

At this point, we've got a nice little .wav playing application:

```
REBOL []

play-sound: func [sound-file] [
    wait 0
    wait-flag: true
```

```
    ring: load sound-file
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
    wait-flag: false
]
wait-flag: false
change-dir %/c/Windows/media
waves: []
foreach file read %. [
    if %.wav = suffix? file [append waves file]
]
view layout [
    vh2 "Click a File to Play:"
    file-list: text-list data waves [
        if wait-flag <> true [
            if error? try [play-sound value] [
                alert "malformed wave"
                close sound-port
                wait-flag: false
            ]
        ]
    ]
    btn "Change Folder" [
        change-dir request-dir
        waves: copy []
        foreach file read %. [
            if %.wav = suffix? file [append waves file]
        ]
        file-list/data: waves
        show file-list
    ]
]
```

This was posted online, and within a few days several readers asked the same question: "How do I get it to play .mp3 files?". REBOL cannot natively play mp3s, so we need to use an external tool to make that happen. Earlier in the tutorial, I included a .dll example that plays mp3 files, but I wanted a slightly more industrial strength solution. I decided to give the well known "LAME" mp3 encoder/decoder a try. I downloaded the compiled Windows version of LAME from http://www.rarewares.org/mp3-lame-bundle.php, and compressed the .exe version of it using the binary resouce embedder found earlier in this tutorial. For the sake of saving space in this tutorial, I uploaded the compressed, embedded code to http://musiclessonz.com/rebol_tutorial/lame.r. The following line writes the lame.exe program to the current directory of your hard drive:

```
do http://musiclessonz.com/rebol_tutorial/lame.r  ; ~250k download
```

To use our media player program without having to download anything, simply put the above lame.r code directly in your script. Once you've got lame.exe on your hard drive, you can use it to convert .mp3 files to .wav files using the format:

```
call/wait {lame.exe --decode your-input.mp3 your-output.wav}
```

I added the line above to my existing program, and changed the "waves" block-building foreach routine to include .mp3 files:

```
waves: []
```

```
foreach file read %. [
    if ((%.wav = suffix? file) or
    (%.mp3 = suffix? file)) [append waves file]
]
```

I also changed the wave playing routine (the action block of the GUI text list), so that if an mp3 file is selected, lame is run and the file is converted to a temporary wav file, and then that wav file is played:

```
file-list: text-list data waves [
    either %.mp3 = suffix? value [
        call/wait rejoin ["lame.exe --decode "
            (to-local-file value) " temp.wav"]
        if wait-flag <> true [
            if error? try [play-sound %temp.wav] [
                alert "malformed wave"
                close sound-port
                wait-flag: false
            ]
        ]
    ] [
        if wait-flag <> true [
            if error? try [play-sound value] [
                alert "malformed wave"
                close sound-port
                wait-flag: false
            ]
        ]
    ]
]
```

With those changes, the code now looks like this:

```
REBOL []

do http://musiclessonz.com/rebol_tutorial/lame.r
play-sound: func [sound-file] [
    wait 0
    wait-flag: true
    ring: load sound-file
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
    wait-flag: false
]
wait-flag: false
change-dir %/c/Windows/media
waves: []
foreach file read %. [
    if ((%.wav = suffix? file) or
        (%.mp3 = suffix? file)) [append waves file]
]
view center-face layout [
    vh2 "Click a File to Play:"
    file-list: text-list data waves [
        either %.mp3 = suffix? value [
            message/text: "Decoding mp3..." show message
            call/wait rejoin ["lame.exe --decode "
```

```
                (to-local-file value) " temp.wav"]
            message/text: "" show message
            if wait-flag <> true [
                if error? try [play-sound %temp.wav] [
                    alert "malformed wave"
                    close sound-port
                    wait-flag: false
                ]
            ]
        ] [
            if wait-flag <> true [
                if error? try [play-sound value] [
                    alert "malformed wave"
                    close sound-port
                    wait-flag: false
                ]
            ]
        ]
    ]
    btn "Change Folder" [
        change-dir request-dir
        waves: copy []
        foreach file read %. [
            if ((%.wav = suffix? file) or
            (%.mp3 = suffix? file)) [append waves file]
        ]
        file-list/data: waves
        show file-list
    ]
    message: h2 "                    "
]
```

That's a cute solution, but the performance is not acceptable for any legitimate use. Large mp3 files take a long time to convert before being played.

Another potential mp3 playing option I considered was a small command line mp3 player called "madplay.exe", available from http://www.rarewares.org/mp3-others.php. Madplay doesn't have any GUI interface - you simply run it on the command line and control playback using keystrokes. Madplay can play many types of media, and keystrokes can be sent to it programatically using the Windows API or the Autoit DLL. I did create a working app using madplay.exe and autoit.dll, but I won't document that code here because it felt like another cobbled together solution.

To find a real solution, I googled "play mp3 dll". The first thing that came up was "libwmp3.dll" from http://www.inet.hr/~zcindori/libwmp3/index.html. Bingo - that did exactly what I wanted. The dll shipped with example usage code written in Visual Basic, C, C++, and Delphi. From these examples, I was able to decipher the required function names, input parameters, and return values, and came up with the following code to access the .dll in REBOL:

```
lib: load/library %libwmp3.dll

Mp3_Initialize: make routine! [
    return: [integer!]
] lib "Mp3_Initialize"

Mp3_OpenFile: make routine! [
    return: [integer!]
    class [integer!]
    filename [string!]
    nWaveBufferLengthMs [integer!]
    nSeekFromStart [integer!]
    nFileSize [integer!]
```

```
] lib "Mp3_OpenFile"

Mp3_Play: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Play"


; The following function and structure aren't required
; to play mp3s, but we'll use them to determine if a
; file is currently being played:

Mp3_GetStatus: make routine! [
    return: [integer!]
    initialized [integer!]
    status [struct! []]
] lib "Mp3_GetStatus"

status: make struct! [
    fPlay [integer!]
    fPause [integer!]
    fStop [integer!]
    fEcho [integer!]
    nSfxMode [integer!]
    fExternalEQ [integer!]
    fInternalEQ [integer!]
    fVocalCut [integer!]
    fChannelMix [integer!]
    fFadeIn [integer!]
    fFadeOut [integer!]
    fInternalVolume [integer!]
    fLoop [integer!]
    fReverse [integer!]
] none

; The following functions stop play and release memory when done:

Mp3_Stop: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Stop"

Mp3_Destroy: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Destroy"
```

Now those functions can be used in REBOL to play any .mp3. It's very easy, using 3 functions:

```
initialized: Mp3_Initialize
Mp3_OpenFile initialized "test.mp3" 1000 0 0
Mp3_Play initialized

; Just change the "test.mp3" file to any .mp3 you want to play.
; Be sure that the file name is sent as a string, and that it's
; written in Windows file format (use the "to-local-file" and
; "what-dir" functions if necessary, to convert from REBOL file
; format).
```

That's all the code required to play an mp3. To check if an mp3 file is currently playing:

```
Mp3_GetStatus initialized status
if ( status/fPlay > 0 ) [print "playing"]
```

To stop an mp3 from playing:

```
Mp3_Stop initialized
```

To clean up afterward:

```
Mp3_Destroy initialized
free lib
```

I added some code to download the libwmp3.dll to the hard drive (of course, the .dll file could also be embedded directly in your code, using the binary resouce embedder from earlier in this tutorial):

```
if not exists? %libwmp3.dll [
    write/binary %libwmp3.dll
    read/binary http://musiclessonz.com/rebol_tutorial/libwmp3.dll
]
```

Now we can add real mp3 playing ability to our little app. Here's the final code:

```
REBOL [title: "Jukebox - Wav/Mp3 Player"]

if not exists? %libwmp3.dll [
    write/binary %libwmp3.dll
    read/binary http://musiclessonz.com/rebol_tutorial/libwmp3.dll
]

lib: load/library %libwmp3.dll

Mp3_Initialize: make routine! [
    return: [integer!]
] lib "Mp3_Initialize"

Mp3_OpenFile: make routine! [
    return: [integer!]
    class [integer!]
    filename [string!]
    nWaveBufferLengthMs [integer!]
    nSeekFromStart [integer!]
    nFileSize [integer!]
] lib "Mp3_OpenFile"

Mp3_Play: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Play"

Mp3_Stop: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Stop"

Mp3_Destroy: make routine! [
```

```
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Destroy"

Mp3_GetStatus: make routine! [
    return: [integer!]
    initialized [integer!]
    status [struct! []]
] lib "Mp3_GetStatus"

status: make struct! [
    fPlay [integer!]
    fPause [integer!]
    fStop [integer!]
    fEcho [integer!]
    nSfxMode [integer!]
    fExternalEQ [integer!]
    fInternalEQ [integer!]
    fVocalCut [integer!]
    fChannelMix [integer!]
    fFadeIn [integer!]
    fFadeOut [integer!]
    fInternalVolume [integer!]
    fLoop [integer!]
    fReverse [integer!]
] none

play-sound: func [sound-file] [
    wait 0
    wait-flag: true
    ring: load sound-file
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
    wait-flag: false
]

wait-flag: false
change-dir %/c/Windows/media
waves: []
foreach file read %. [
    if ((%.wav = suffix? file) or
        (%.mp3 = suffix? file)) [append waves file]
]

initialized: Mp3_Initialize

view center-face layout [
    vh2 "Click a File to Play:"
    file-list: text-list data waves [
        Mp3_GetStatus initialized status
        either %.mp3 = suffix? value [
            if (wait-flag <> true) and (status/fPlay = 0) [
                file: rejoin [to-local-file what-dir "\" value]
                Mp3_OpenFile initialized file 1000 0 0
                Mp3_Play initialized
            ]
        ] [
            if (wait-flag <> true) and (status/fPlay = 0) [
                if error? try [play-sound value] [
                    alert "malformed wave"
                    close sound-port
```

```
                    wait-flag: false
                ]
            ]
        ]
    ]
    across
    btn "Change Folder" [
        change-dir request-dir
        waves: copy []
        foreach file read %. [
            if ((%.wav = suffix? file) or
            (%.mp3 = suffix? file)) [append waves file]
        ]
        file-list/data: waves
        show file-list
    ]
    btn "Stop" [
         close sound-port
         wait-flag: false
         if (status/fPlay > 0) [Mp3_Stop initialized]
    ]
]

Mp3_Destroy initialized
free lib
```

I wrote a longer example that shows how to use other features of the libwmp3.dll: pause/resume, volume control, fast forward/rewind, looping, reverse play and vocal removal. It's available at http://www.rebol.org/view-script.r?script=mp3-player-libwmp.r. The function prototypes in that example demonstrate how to use all the other functions in the library: equalizer settings, stream playing, retrieval of ID field and recorded data info, effect application (echo, reverb, etc.), and more. The example below demonstrates how to attach volume, loop play, and seek parameters to GUI slide controls (this example only works with MP3 files):

```
REBOL [Title: "Jukebox"]

if not exists? %libwmp3.dll [
    write/binary %libwmp3.dll
    read/binary http://musiclessonz.com/rebol_tutorial/libwmp3.dll
]
lib: load/library %libwmp3.dll
Mp3_Initialize: make routine! [
    return: [integer!]
] lib "Mp3_Initialize"
Mp3_OpenFile: make routine! [
    return: [integer!]
    class [integer!]
    filename [string!]
    nWaveBufferLengthMs [integer!]
    nSeekFromStart [integer!]
    nFileSize [integer!]
] lib "Mp3_OpenFile"
Mp3_Play: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Play"
Mp3_Stop: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Stop"
Mp3_Destroy: make routine! [
```

```
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Destroy"
Mp3_GetStatus: make routine! [
    return: [integer!]
    initialized [integer!]
    status [struct! []]
] lib "Mp3_GetStatus"
status: make struct! [
    fPlay [integer!]
    fPause [integer!]
    fStop [integer!]
    fEcho [integer!]
    nSfxMode [integer!]
    fExternalEQ [integer!]
    fInternalEQ [integer!]
    fVocalCut [integer!]
    fChannelMix [integer!]
    fFadeIn [integer!]
    fFadeOut [integer!]
    fInternalVolume [integer!]
    fLoop [integer!]
    fReverse [integer!]
] none
Mp3_Time: make struct! [
    ms [integer!]
    sec [integer!]
    bytes [integer!]
    frames [integer!]
    hms_hour [integer!]
    hms_minute [integer!]
    hms_second [integer!]
    hms_millisecond [integer!]
] none
TIME_FORMAT_SEC: 2
SONG_BEGIN: 1
SONG_CURRENT_FORWARD: 4
Mp3_Seek: make routine! [
    return: [integer!]
    initialized [integer!]
    fFormat [integer!]
    pTime [struct! []]
    nMoveMethod [integer!]
] lib "Mp3_Seek"
Mp3_PlayLoop: make routine! [
    return: [integer!]
    initialized [integer!]
    fFormatStartTime [integer!]
    pStartTime [struct! []]
    fFormatEndTime [integer!]
    pEndTime [struct! []]
    nNumOfRepeat [integer!]
] lib "Mp3_PlayLoop"
Mp3_GetSongLength: make routine! [
    return: [integer!]
    initialized [integer!]
    pLength [struct! []]
] lib "Mp3_GetSongLength"
Mp3_GetPosition: make routine! [
    return: [integer!]
    initialized [integer!]
    pTime [struct! []]
] lib "Mp3_GetPosition"
```

```
Mp3_SetVolume: make routine! [
    return: [integer!]
    initialized [integer!]
    nLeftVolume [integer!]
    nRightVolume [integer!]
] lib "Mp3_SetVolume"
Mp3_GetVolume: [
    initialized [integer!]
    pnLeftVolume [integer!]
    pnRightVolume [integer!]
    return: [integer!]
] lib "Mp3_GetVolume"
Mp3_VocalCut: make routine! [
    return: [integer!]
    initialized [integer!]
    fEnable [integer!]
] lib "Mp3_VocalCut"
Mp3_ReverseMode: make routine! [
    return: [integer!]
    initialized [integer!]
    fEnable [integer!]
] lib "Mp3_ReverseMode"
Mp3_Close: make routine! [
    return: [integer!]
    initialized [integer!]
] lib "Mp3_Close"
waves: []
foreach file read %. [
    if (%.mp3 = suffix? file) [append waves file]
]
append waves "(CHANGE FOLDER...)"
initialized: Mp3_Initialize
view center-face layout [
    vh2 "Click a File to Play:"
    file-list: text-list data waves [
        if value = "(CHANGE FOLDER...)" [
            new-dir: request-dir
            if new-dir = none [break]
            change-dir new-dir
            waves: copy []
            foreach file read %. [
                if (%.mp3 = suffix? file) [append waves file]
            ]
            append waves "(CHANGE FOLDER...)"
            file-list/data: waves
            show file-list
            break
        ]
        Mp3_GetStatus initialized status
        if (status/fPlay = 0) [
            file: rejoin [to-local-file what-dir "\" value]
            Mp3_OpenFile initialized file 1000 0 0
            Mp3_Play initialized
        ]
    ]
    across
    tabs 40
    text "Seek:   "
    tab slider 140x15 [
        plength: make struct! Mp3_Time compose [0 0 0 0 0 0 0 0]
        Mp3_GetSongLength initialized plength
        location: to-integer (value * plength/sec)
        ptime: make struct! Mp3_Time compose [0 (location) 0 0 0 0 0 0]
```

```
        Mp3_Seek initialized TIME_FORMAT_SEC ptime SONG_BEGIN
        Mp3_Play initialized
    ]
    return
    text "Volume: "
    tab slider 140x15 [
        volume: to-integer value * 100
        Mp3_SetVolume initialized volume volume
    ]
    return
    btn "Reverse" [
        Mp3_GetStatus initialized status
        either (status/fReverse > 0) [
            Mp3_ReverseMode initialized 0
        ] [
            Mp3_ReverseMode initialized 1
        ]
    ]
    btn "Vocal-Cut" [
        Mp3_GetStatus initialized status
        either (status/fVocalCut > 0) [
            Mp3_VocalCut initialized 0
        ] [
            Mp3_VocalCut initialized 1
        ]
    ]
    return
    tabs 50
    text "Loop Start:"
    tab start-slider: slider 120x15 []
    return
    text "Loop End:  "
    tab end-slider: slider 120x15 []
    return
    btn "Play Loop" [
        plength: make struct! Mp3_Time compose [0 0 0 0 0 0 0 0]
        Mp3_GetSongLength initialized plength
        s-loc: to-integer (start-slider/data * plength/sec)
        pStartTime: make struct! Mp3_Time compose [0 (s-loc) 0 0 0 0 0 0]
        end-loc: to-integer (end-slider/data * plength/sec)
        pEndTime: make struct! Mp3_Time compose [0 (end-loc) 0 0 0 0 0 0]
        ; TIME_FORMAT_SEC: 2
        Mp3_PlayLoop initialized 2 pStartTime 2 pEndTime 1000  ; 1000x
    ]
    btn 58 "Stop" [
        Mp3_GetStatus initialized status
        if (status/fPlay > 0) [Mp3_Stop initialized]
    ]
]
Mp3_Destroy initialized
free lib
```

Libwmp3.dll is a very powerful and easy solution for playing mp3 files in any Windows programming language. If you're interested in playing mp3s in REBOL, it's a must-have.

10.15 Case 15 - Creating the REBOL "Demo"

At the beginning of this tutorial, a short application was provided to demonstrate how potent REBOL code can be. The 10 programs included in that demo are all shortened versions of other pieces of code found throughout this tutorial:

The "paint" program was covered in the section of the tutorial about the draw dialect:

```
view center-face layout [
    s: area black 650x350 feel [
        engage: func [f a e] [
            if a = 'over [
                append s/effect/draw e/offset
                show s
            ]
            if a = 'up [append s/effect/draw 'line]
        ]
    ] effect [draw [line]]
    b: btn "Save" [
        save/png %a.png to-image s
        alert "Saved 'a.png'"
    ]
    btn "Clear" [
        s/effect/draw: copy [line]
        show s]
    ]
]
```

The "game" is the obfuscated snake program covered earlier:

```
do[p: :append u: :reduce k: :pick r: :random y: :layout q: 'image z: :if
g: :to-image v: :length? x: does[alert join{SCORE: }[v b]quit]s: g y/tight
[btn red 10x10]o: g y/tight[btn tan 10x10]d: 0x10 w: 0 r/seed now b: u[q
o(((r 19x19)* 10)+ 50x50)q s(((r 19x19)* 10)+ 50x50)]view center-face
y/tight[c: area 305x305 effect[draw b]rate 15 feel[engage: func[f a e][z a
= 'key[d: select u['up 0x-10 'down 0x10 'left -10x0 'right 10x0]e/key]z a
= 'time[z any[b/6/1 < 0 b/6/2 < 0 b/6/1 > 290 b/6/2 > 290][x]z find(at b
7)b/6[x]z within? b/6 b/3 10x10[p b u[q s(last b)]w: 1 b/3:((r 29x29)*
10)]n: copy/part b 5 p n(b/6 + d)for i 7(v b)1 [either(type?(k b i)=
pair!)[p n k b(i - 3)][p n k b i]]z w = 1[clear(back tail n)p n(last b)w:
0]b: copy n show c]]]do[focus c]]]
```

The "puzzle" is the tile program explained in the first section of the tutorial about GUIs:

```
alert {Arrange tiles alphabetically:}
view center-face layout [
    origin 0x0 space 0x0 across
    style p button 60x60 [
        if not find [0x60 60x0 0x-60 -60x0] face/offset - x/offset [exit]
        temp: face/offset face/offset: x/offset x/offset: temp
    ]
    p "O" p "N" p "M" p "L" return
    p "K" p "J" p "I" p "H" return
    p "G" p "F" p "E" p "D" return
    p "C" p "B" p "A" x: p white edge [size: 0]
]
```

The "calendar" is a simple application in which the user selects a day using the date requester

function. Events for the day are typed into an area widget and then appended to a text file. The text file is searched every time a date is chosen. If the chosen date is found, the events for that day are shown in the area widget, which can be edited and saved back to the text file:

```
do the-calendar: [
    if not (exists? %s) [write %s ""]
    the-date: request-date
    view center-face layout [
        h5 to-string the-date
        aa: area to-string select to-block (
            find/last (to-block read %s) the-date
        ) the-date
        btn "Save" [
            write/append %s rejoin [the-date " {" aa/text "} " ]
            unview
            do the-calendar
        ]
    ]
]
```

The "video" program was covered in the section of the tutorial about multitasking. All it does is continually load and display images from a web cam server. The image refresh is handled using a feel-engage loop, which checks for a timer event:

```
video-address: to-url request-text/title/default "URL:" trim {
    http://tinyurl.com/m54ltm}
view center-face layout [
    image load video-address 640x480 rate 0 feel [
        engage: func [f a e] [
            if a = 'time [
                f/image: load video-address
                show f
            ]
        ]
    ]
 ]
```

The "IP" program was covered in the section about the REBOL parse dialect. This program reads a web page which displays the remote WAN IP address of the user's computer in the title tag, then parses out all the extra text and displays the IP address, along with the user's local IP address (the local address is gotten by using REBOL's built in dns:// protocol:

```
parse read to-url "http://guitarz.org/ip.cgi" [
    thru <title> copy my to </title>
]
i: last parse my none
alert to-string rejoin [
    "WAN: " i " -- LAN: " read join dns:// read dns://
]
```

The "email" program is extremely simple. The user enters email account information into a GUI text field, and then the mail from that account is read using REBOL's native POP protocol. The contents of the mailbox are displayed in REBOL's built-in text editor, each separated by 6 newlines:

```
view center-face layout [
    email-login: field "pop://user:pass@site.com"
    btn "Read" [
```

```
        my-mail: copy []
        foreach i (read to-url email-login/text) [
            append my-mail join i "^/^/^/^/^/^/"
            editor my-mail
        ]
    ]
]
```

The "days between" program was covered in an earlier case study. Here's a simple version of the program (an example given in the first part of the case study):

```
view center-face layout [
    btn "Start" [sd: request-date]
    btn "End" [
        ed: request-date
        db/text: to-string (ed - sd)
        show db
    ]
    text "Days Between:"
    db: field
]
```

The "sounds" program was also covered earlier:

```
play-sound: func [sound-file] [
    wait 0 ring: load sound-file
    wait-flag: 1
    sound-port: open sound://
    insert sound-port ring
    wait sound-port
    close sound-port
    wait-flag: 0
]
wait-flag: 0
change-dir %/c/Windows/media
do get-waves: [
    waves-list: copy []
    foreach i read %. [
        if %.wav = suffix? i [
            append waves-list i
        ]
    ]
]
view center-face layout [
    waves-gui-list: text-list data waves-list [
        if wait-flag <> 1 [
            if error? try [play-sound value] [
                alert "Error"
                close sound-port
                wait-flag: 0
            ]
        ]
    ]
    btn "Dir" [
        change-dir request-dir
        do get-waves
        waves-gui-list/data: waves-list
        show waves-gui-list
    ]
```

```
]
```

The "FTP" program is a stripped down version of the "FTP Tool" explained earlier:

```
view center-face layout [
    px: field "ftp://user:pass@site.com/folder/" [
        either dir? to-url value [
            f/data: sort read to-url value
            show f
        ][
            editor to-url value
        ]
    ]
    f: text-list [
        editor to-url join px/text value
    ]
    btn "?" [
        alert {
            Type a URL path to browse (nonexistent files are created).
            Click files to edit.
        }
    ]
]
```

I enclosed all of those examples in a simple GUI, with buttons to run each program:

```
REBOL [title: "Demo"]

view layout [
    style h btn 150
    h "Paint" [
        ; code for the paint program goes here
    ]
    h "Game" [
        ; code for the game program goes here
    ]
    h "Puzzle" [
        ; code for the puzzle program goes here
    ]
    h "Calendar" [
        ; code for the calendar program goes here
    ]
    h "Video" [
        ; code for the video program goes here
    ]
    h "IPs" [
        ; code for the IP program goes here
    ]
    h "Email" [
        ; code for the email program goes here
    ]
    h "Days" [
        ; code for the days-between program goes here
    ]
    h "Sounds" [
        ; code for the sound program goes here
    ]
    h "FTP" [
        ; code for the FTP program goes here
    ]
```

```
]
```

To make the demo as compact as possible, I used the same techniques as in the obfuscated snake program (from earlier in the tutorial). Here are the global functions that I renamed with shorter word labels:

```
p: :append kk: :pick r: :random y: :layout q: 'image z: :if gg: :to-image
v: :length? g: :view k: :center-face ts: :to-string tu: :to-url sh: :show
al: :alert rr: :request-date co: :copy
```

I also renamed other functions within their local contexts. In the following code, the value "s/effect /draw" is assigned the variable "pk". That saves having to write "s/effect/draw" again. That value does not exist in the global context, so that variable must be assigned locally. This type of shortened local variable assignment occurs several times throughout the demo code:

```
view layout [
    s: area black 650x350 feel [
        engage: func [f a e] [
            if a = 'over [
                append pk: s/effect/draw e/offset
                show s
            ]
            if a = 'up [append pk 'line]
        ]
    ] effect [
        draw [line]
    ]
]
```

To finish the application, I simply removed any spaces which surrounded parentheses or brackets. The final code is found in the first section of this tutorial - it's less than 1/2 a page of printed code:

10.16 Case 16 - Guitar Chord Chart Printer

This program was written to help students in high school jazz band quickly play all of the common extended, altered, and complex chord types. It creates and prints instant guitar chord diagram charts for songs. Although it was written to help teach complex jazz chords, it can also be used to create chord charts for any other type of music (with simpler chords): folk, rock, blues, pop, etc.

When I set out to create this program, here's what I envisioned:

1. Users should be able to select from a list of root notes (A, Bb, C#, etc.), and sonorities (major, minor, 7(#5b9), etc.) for each chord in a song.
2. A list of selected chords should be shown in a text area.
3. When the user has added all the chords in a song, they should be able to click a button to view the chords in their browser. By displaying in a browser, the user can adjust printer settings to print charts at different sizes.
4. I wanted the user to be able to save and load chord lists to a text file, and to be able to create a zip file of all the rendered chords in a song (the HTML and all the images used to display the song on the browser).

To understand how this program works, it's essential to understand some basics about how chords are formed on the guitar fretboard. Every chord label in our musical system has two parts: a root note (letter name), and a sonority (type/characteristic sound). The traditional way to teach chord theory on guitar is to use two fretboard fingering patterns: 1 with the root note of the chord on the 6th string, and another with the root note on the 5th string. Each shape is a diagram of "intervals" (notes from a major scale, "do re mi fa so la ti do") that are combined to form chords. Every type of chord is made up of a specific formula of intervals, and that unique combination of intervals creates a predictable characteristic sound. The shapes can be slid up or down along the fretboard, so that the root note (interval number 1) in the diagram is placed on the specified root note of a given chord.

Here are the interval fretboard diagrams, showing where to put your fingers to create chords (These are basically pictures of the fretboard, as if the guitar is sitting upright in front of you. Search for "how to read guitar chord diagrams" in Google to understand more about how fretboard diagrams work.):

```
Root 6 interval shapes:          Root 5 interval shapes:
_____________                    _____________
| | | | 4 |                      | | | | | |
| 3 6 9 | 7                      | | | | | |
1 | | | 5 1                      | | | | 1 4
| | 7 3 | |                      | | 3 6 | |
| 5 1 4 6 9                      5 1 4 | 9 5
| | | | | |                      | | | 7 | |
| | 9 | 7 |                      | | 5 1 3 6
```

Here are the interval patterns used to create chords, along with the various symbols seen in music to represent each type of chord:

```
CHORD TYPE:           INTERVALS:           SYMBOLS:

Power Chord           1    5                5
Major Triad           1    3    5           none  (just a root noot)
Minor Triad           1   b3    5           m, min, mi, -
Dominant 7            1    3   (5)  b7      7
Major 7               1    3   (5)   7      maj7, M7, (triangle) 7
Minor 7               1   b3   (5)  b7      m7, min7, mi7, -7
Half Diminished 7     1   b3   b5   b7      m7b5, (circle with line) 7
Diminished 7          1   b3   b5  bb7 (6)  dim7, (circle) 7
Augmented 7           1    3   #5   b7      7aug, 7(#5), 7(+5)

Add these intervals to the above 7th chords to create extended chords:
```

```
9 (is same as 2)   11 (is same as 4)   13 (is same as 6)

Examples:              9          =    1   3  (5)  b7    9
                       min9       =    1  b3  (5)  b7    9
                       13         =    1   3   5   b7   13
                       9(+5)      =    1   3  #5   b7    9
                       maj9(#11)  =    1   3  (5)   7    9  #11

Here are some more common chord types:

"sus"       =  change 3 to 4
"sus2"      =  change 3 to 2
"add9"      =  1 3 5 9  (same as "add2", there's no 7 in "add" chords)
"6,  maj6"  =  1 3 5 6
"m6, min6"  =  1 b3 5 6
"6/9"       =  1 3 5 6 9
11          =  1 b7 9 11
"/"         =  Bassist plays the note after the slash

NOTE:  When playing complex chords (jazz chords) in a band setting,
guitarists typically SHOULD NOT PLAY THE ROOT NOTE of the chord
(the bassist or keyboardist will play it).  In diagrams created by
this program, unnecessary notes will be indicated by light circles,
and required notes will be indicated by dark circles.
```

Here are the locations of root notes on the 6th and 5th strings:

```
6th string notes:               5th string notes:

0  1  3  5  7  8  10  12        0  2  3  5  7  8  10  12
E  F  G  A  B  C  D   E        A  B  C  D  E  F  G   A

The sharp symbol ("#") moves notes UP    one fret
The flat  symbol ("b") moves notes DOWN  one fret
```

Here's my plan of attack to create the program:

1. Use a text list widget to display a clickable list of all possible root notes.
2. Use a text list widget to display a clickable list of all possible chord types.
3. Use an area widget to display the chords chosen by the user (root note + chord type). Every time the user clicks a new chord, rejoin the existing text with the newly chosen chord text. Put each new chord label on a new line.
4. Add a button to let the user select when to render and display images of all the chords in the browser.
5. To create the images, I'll use REBOL's built in draw dialect. I'll also create a simple HTML page to display the rendered images, and save all those files to a newly created subdirectory. Then launch the browser to view the created page.
6. To draw the images, first I'll draw a grid of lines, each 20 pixels apart. Vertical lines represent strings, horizontal lines represent frets.
7. I'll plot circles onto the above grid, where the required intervals are located in every selected chord type. To do that, I'll create a map of all the possible *chord types*, telling the program which interval numbers are required to create each selected chord type. I'll also create a map of all the possible *interval numbers*, telling the program where to plot each required interval (XxY coordinate) listed in the chord type map. Finally, I'll create a map of the frets at which all possible root notes are located. The program will simply read the chord labels entered by the user, plot the required circles at the appropriate coordinates, for all intervals required in each chord. I'll also print the appropriate fret number for the given root note, and chord label too, directly in each image.
8. I'll need to create separate chord type, coordinate, and root note maps for both the 6th and 5th

string shapes. I'll need to run through the rendering process twice for every chord (once for the 6th string shape and once for the 5th string shape).

I started by creating all the required maps. Having those ready would help me deal more concretely with the action code. The root note maps are just blocks containing all possible root notes, followed by the frets at which they're found. The interval maps are simply blocks containing every interval name, followed by the coordinates at which they should be plotted on the grid diagram. The shape maps are simply blocks containing:

1. A chord label (symbol) used to indicate each specific chord type.
2. A list of various other names and labels used to represent each chord type (all contained in a string).
3. A block of the intervals used to create each chord type. Because several intervals can be found multiple places in each diagram, I included some numbers multiple times, using multiple labels (i.e., the root note is seen as 1, 11, and 111 in various chords).

Here are all the root 6 maps:

```
root6-shapes: [
    "." "major triad, no symbol (just a root note)" [1 3 5 11 55 111]
    "m" "minor triad, min, mi, m, -" [1 b3 5 11 55 111]
    "aug" "augmented triad, aug, #5, +5" [1 3 b6 11 111]
    "dim" "diminished triad, dim, b5, -5" [1 b3 b5 11]
    "5" "power chord, 5" [1 55]
    "sus4" "sus4, sus" [1 4 5 11 55 111]
    "sus2" "sus2, 2" [1 99 5 11]
    "6" "major 6, maj6, ma6, 6" [1 3 5 6 11]
    "m6" "minor 6, min6, mi6, m6" [1 b3 5 6 11]
    "69" "major 6/9, 6/9, add6/9" [1 111 3 13 9]
    "maj7" "major 7, maj7, ma7, M7, (triangle) 7" [1 3 5 7 11 55]
    "7" "dominant 7, 7" [1 3 5 b7 11 55]
    "m7" "minor 7, min7, mi7, m7, -7" [1 b3 5 b7 11 55]
    "m7(b5)" "half diminished, min7(b5), (circle w/ line), m7(-5), -7(b5)"
        [1 b3 b5 b7 11]
    "dim7" "diminished 7, dim7, (circle) 7" [1 b3 b5 6 11]
    "7sus4" "dominant 7 sus4 (7sus4)" [1 4 5 b7 55 11]
    "7sus2" "dominant 7 sus2 (7sus2)" [1 b7 99 5 11]
    "7(b5)" "dominant 7 flat 5, 7(b5), 7(-5)" [1 3 b5 b7 11]
    "7(+5)" "augmented 7, 7(#5), 7(+5)" [1 3 b6 b7 11]
    "7(b9)" "dominant 7 flat 9, 7(b9), 7(-9)" [1 3 5 b7 b9]
    "7(+9)" "dominant 7 sharp 9, 7(#9), 7(+9)" [1 111 3 b77 b33]
    "7(b5b9)" "dominant 7 b5 b9, 7(b5b9), 7(-5-9)" [1 3 b5 b7 b9]
    "7(b5+9)" "dominant 7 b5 #9, 7(b5#9), 7(-5+9)" [1 3 b5 b7 b33]
    "7(+5b9)" "augmented 7 flat 9, aug7(b9), 7(#5b9)" [1 3 b6 b7 b9]
    "7(+5+9)" "augmented 7 sharp 9, aug7(#9), 7(#5#9)" [1 3 b6 b7 b33]
    "add9" "add9, add2" [1 3 5 999 55 11]
    "madd9" "minor add9, min add9, m add9, m add2" [1 b3 5 999 55 11]
    "maj9" "major 9, maj9, ma9, M9, (triangle) 9" [1 3 5 7 9]
    "maj9(+11)" "major 9 sharp 11, maj9(#11), M9(+11)" [1 3 7 9 b5]
    "9" "dominant 9, 9" [1 3 5 b7 9 55]
    "9sus" "dominant 9 sus4, 9sus4, 9sus" [1 4 5 b7 9 55]
    "9(+11)" "dominant 9 sharp 11, 9(#11), 9(+11)" [1 3 b7 9 b5]
    "m9" "minor 9, min9, mi9, m9, -9" [1 b3 5 b7 9 55]
    "11" "dominant 11, 11" [1 b7 99 44 11]
    "maj13" "major 13, maj13, ma13, M13, (triangle) 13" [1 3 55 7 11 13]
    "13" "dominant 13, 13" [1 3 55 b7 11 13]
    "m13" "minor 13, min13, mi13, m13, -13" [1 b3 55 b7 11 13]
]
root6-map:  [
    1 20x70 11 120x70 111 60x110 3 80x90 33 40x50 b3 80x70 5 100x70
    55 40x110 b5 100x50 7 60x90 b7 60x70 9 120x110 99 80x50 6 60x50
    13 100x110 4 80x110 44 100x30 999 60x150 b77 100x130 b33 120x130
```

```
    b9 120x90 b6 100x90 b55 40x90
]
root6-notes:  [
    "e" {12} "f" {1} "f#" {2} "gb" {2} "g" {3} "g#" {4} "ab" {4}
    "a" {5} "a#" {6} "bb" {6} "b" {7} "c" {8} "c#" {9} "db" {9} "d" {10}
    "d#" {11} "eb" {11}
]
```

The root 5 maps simply mirror the lists above, with values altered for the 5th string:

```
root5-shapes: [
    "." "major triad, no symbol (just a root note)" [1 3 5 11 55]
    "m" "minor triad, min, mi, m, -" [1 b3 5 11 55]
    "aug" "augmented triad, aug, #5, +5" [1 3 b6 11 b66]
    "dim" "diminished triad, dim, b5, -5" [1 b3 b5 11]
    "5" "power chord, 5" [1 55]
    "sus4" "sus4, sus" [1 4 5 11 55]
    "sus2" "sus2, 2" [1 9 5 11 55]
    "6" "major 6, maj6, ma6, 6" [1 3 55 13 11]
    "m6" "minor 6, min6, mi6, m6" [1 b3 55 13 11]
    "69" "major 6/9, 6/9, add6/9" [1 33 6 9 5]
    "maj7" "major 7, maj7, ma7, M7, (triangle) 7" [1 3 5 7 55]
    "7" "dominant 7, 7" [1 3 5 b7 55]
    "m7" "minor 7, min7, mi7, m7, -7" [1 b3 5 b7 55]
    "m7(b5)" "half diminished, min7(b5), (circle w/ line), m7(-5), -7(b5)"
        [1 b3 b5 b7 b55]
    "dim7" "diminished 7, dim7, (circle) 7" [1 b33 b5 6 111]
    "7sus4" "dominant 7 sus4, 7sus4" [1 4 5 b7 55]
    "7sus2" "dominant 7 sus2, 7sus2" [1 9 5 b7 55]
    "7(b5)" "dominant 7 flat 5, 7(b5), 7(-5)" [1 33 b5 b7 111]
    "7(+5)" "augmented 7, 7(#5), 7(+5)" [1 33 b6 b7 111]
    "7(b9)" "dominant 7 flat 9, 7(b9), 7(-9)" [1 33 5 b7 b9]
    "7(+9)" "dominant 7 sharp 9, 7(#9), 7(+9)" [1 33 b7 b3]
    "7(b5b9)" "dominant 7 b5 b9, 7(b5b9), 7(-5-9)" [1 33 b5 b7 b9]
    "7(b5+9)" "dominant 7 b5 #9, 7(b5#9), 7(-5+9)" [1 33 b5 b7 b3]
    "7(+5b9)" "augmented 7 flat 9, aug7(b9), 7(#5b9)" [1 33 b6 b7 b9]
    "7(+5+9)" "augmented 7 sharp 9, aug7(#9), 7(#5#9)" [1 33 b7 b3 b6]
    "add9" "major add9, add9, add2" [1 3 5 99 55]
    "madd9" "minor add9, min add9, m add9, m add2" [1 b3 5 99 55]
    "maj9" "major 7, maj9, ma9, M9, (triangle) 9" [1 33 5 7 9]
    "maj9(+11)" "major 9 sharp 11, maj9(#11), M9(+11)" [1 33 b5 7 9]
    "9" "dominant 9, 9" [1 33 5 b7 9]
    "9sus" "dominant 9 sus4, 9sus4, 9sus" [1 44 5 b7 9]
    "9(+11)" "dominant 9 sharp 11, 9(#11), 9(+11)" [1 33 b5 b7 9]
    "m9" "minor 9, min9, mi9, m9, -9" [1 b33 5 b7 9]
    "11" "dominant 11, 11" [1 b7 9 44 444]
    "maj13" "major 13, maj13, ma13, M13, (triangle) 13" [1 3 55 7 13]
    "13" "dominant 13, 13" [1 3 55 b7 13]
    "m13" "minor 13, min13, mi13, m13, -13" [1 b3 55 b7 13]
]
root5-map:  [
    1 40x70 11 80x110 111 100x30 3 100x110 33 60x50 b33 60x30 5 120x70
    55 60x110 b5 120x50 7 80x90 b7 80x70 9 100x70 6 80x50 13 120x110
    4 100x130 44 60x70 444 120x30 99 80x150 b3 100x90 b9 100x50 b6 120x90
    b66 60x130 b55 60x90
]
root5-notes: [
    "a" {12} "a#" {1} "bb" {1} "b" {2} "c" {3} "c#" {4} "db" {4}
    "d" {5} "d#" {6} "eb" {6} "e" {7} "f" {8} "f#" {9} "gb" {9} "g" {10}
    "g#" {11} "ab" {11}
]
```

Next, I wrote the code to draw the grid image, onto which all the circles will be plotted. This was simply a matter of creating 2 blocks of line start/end points (one block each for vertical and horizontal lines), drawing those lines onto a display, and then saving that display as an image:

```
f: copy []
for n 20 160 20 [append f reduce ['line (as-pair 20 n) (as-pair 120 n)]]
for n 20 120 20 [append f reduce ['line (as-pair n 20) (as-pair n 160)]]
fretboard: to-image layout/tight [box white 150x180 effect [draw f]]
```

To begin with the action code, I created the GUI skeleton code for the program. Here's the layout, based on the specs I came up with earlier (I also added a button for help):

```
view center-face layout [
    across
    t1: text-list 60x270 data [
        "E" "F" "F#" "Gb" "G" "G#" "Ab" "A" "A#" "Bb" "B" "C" "C#" "Db"
        "D" "D#" "Eb"
    ]
    ; The chord label list is simply extracted from the block above:
    t2: text-list 330x270 data extract/index root6-shapes 3 2 []
    return
    a: area
    return
    btn "Create Chart" []
    btn "Save" []
    btn "Load" []
    btn "Create Zip" []
    btn "Help" [editor help]  ; help will just be a long string of text
]
```

I wanted to have the chord labels added to the text area when the user selected a chord type. Here's the code I came up with:

```
t2: text-list 330x270 data extract/index root6-shapes 3 2 [

    ; When a chord type is clicked, do this:

    either empty? a/text [

        ; If the text area is empty, insert the root note and chord
        ; type into the text area:

        a/text: rejoin [
            copy t1/picked " "
            pick root6-shapes ((index? find root6-shapes value) - 1)
        ]
    ] [

        ; If the text area is not empty, rejoin the existing text, a
        ; newline, and the root note and chord type together.

        a/text: rejoin [
            a/text newline copy t1/picked " "
            pick root6-shapes ((index? find root6-shapes value) - 1)
        ]
    ]

    ; Display the added chord:
```

```
    show a

]
```

Now I need to write the code to actually create the diagrams for each chord in the list. I put that code directly in the action block of the "Create Chart" button:

```
btn "Create Chart" [if error? try [

    ; Create a chords subdirectory if it doesn't exist:

    make-dir %chords

    ; Erase the temporary contents each time;

    delete/any %chords/*.*

    ; Start creating the HTML layout:

    html: copy "<html><body bgcolor=#ffffffff>"

    ; Loop through each chord in the text list:

    foreach [root spacer1 spacer2 type] (parse/all form a/text " ") [

        ; Start creating a draw block, which contains the fretboard image
        ; we created earlier:

        diagram: copy [image fretboard]
        diagram2: copy [image fretboard]

        ; This is the loop that plots each of the intervals in the chord
        ; formula block:

        root1: copy root
        foreach itvl (third find root6-shapes type) [
            either find [1 55] itvl [

                ; Plot a white circle for intervals 1 and 55:

                append diagram reduce [
                    'fill-pen white 'circle (select root6-map itvl) 5
                ]
            ] [

                ; Plot a black circle for all other intervals:

                append diagram reduce [
                    'fill-pen black 'circle (select root6-map itvl) 5
                ]
            ]
        ]

        ; Plot the root note and fret number text in the chord image:

        append diagram reduce ['text (trim/all join root1 type) 20x0]
        append diagram reduce [
            'text
            trim/all to-string (
                select root6-notes trim/all to-string root1
            )
```

```
            130x65
        ]

        ; Render the collected draw block and save the created image to
        ; a file:

        save/png
            to-file trim/all rejoin [
                %./chords/ (replace/all root1 {#} {sharp}) type ".png"
            ]
            to-image layout/tight [
            box white 150x180 effect [draw diagram]
        ]

        ; Add code to our HTML string to display the created image:

        append html rejoin [
            {<img src="./}
            trim/all rejoin [
                replace/all copy root1 {#} {sharp} type ".png"
            ]
            {">}
        ]

        ; Do the entire process above again to create a root 5 image:

        foreach itvl (third find root5-shapes type) [
            either find [1] itvl [
                append diagram2 reduce [
                    'fill-pen white 'circle (select root5-map itvl) 5
                ]
            ] [
                append diagram2 reduce [
                    'fill-pen black 'circle (select root5-map itvl) 5
                ]
            ]
        ]
        append diagram2 reduce ['text (trim/all join root type) 20x0]
        append diagram2 reduce [
            'text
            trim/all to-string (
                select root5-notes trim/all to-string root
            )
            130x65
        ]
        save/png
            to-file trim/all rejoin [
                %./chords/ (replace/all root {#} {sharp})
                type "5th.png"
            ]
            to-image layout/tight [
            box white 150x180 effect [draw diagram2]
        ]
        append html rejoin [
            {<img src="./} (trim/all rejoin [
                replace/all root {#} {sharp} type "5th.png"
            ]) {">}
        ]
    ]

    ; Finish up the HTML code, save it to a file, and display it in
    ; the user's browser:
```

```
    append html [</body></html>]
    write %./chords/chords.html trim/auto html
    browse %./chords/chords.html
] [

    ; If there was an error anywhere in the rendering process (because
    ; the user incorrectly edited chord labels), alert them to make
    ; changes:

    alert "Error - please remove improper chord labels."

]]
btn "Save" [

    ; Save the selected chords to a text file, with an error check to
    ; avoid accidently overwriting existing files:

    savefile: to-file request-file/file/save %/c/mysong.txt
    if exists? savefile [
        alert "Please choose a file name that does not already exist."
        return
    ]
    if error? try [save savefile a/text] [alert "File not saved"]
]
btn "Load" [

    ; Load a saved chord list and display it in the text area.  The
    ; error check is there in case the user clicks the "cancel" button:

    if error? try [
        a/text: load to-file request-file/file %/c/mysong.txt
        show a
    ] []
]

btn "Create Zip" [

    ; This routine creates a zip file of the created HTML file and every
    ; rendered image in the ./chords folder.  It uses rebzip by Vincent
    ; Ecuyer:

    if not exists? %chords/ [alert "Create A Chart First" return]
    do to-string to-binary decompress 64#{

        ; Insert here the compressed zebzip code by Vincent Ecuyer,
        ; found at http://www.rebol.org/view-script.r?script=rebzip.r

    }
    zipfile: to-file request-file/file/save %/c/mysong.zip
    if exists? zipfile [
        alert "Please choose a file name that does not already exist."
        return
    ]
    zip/deep zipfile %chords/
]
btn "Help" [editor help]
```

Here's the final program, with the help text and rebzip code included:

```
REBOL [title: "Guitar Chords"]
```

```
help: {
    This program creates guitar chord diagram charts for songs.  It was
    written to help students in high school jazz band quickly play all of
    the common extended, altered, and complex chord types.  It can also
    be used to create chord charts for any other type of music (with
    simpler chords):  folk, rock, blues, pop, etc.

    To select chords for your song, click the root note (letter name:  A,
    Bb, C#, etc.), and then the sonority (major, minor, 7(#5b9), etc.) of
    each chord.  The list of chords you've selected will be shown in the
    text area below.  When you've added all the chords needed to play your
    song, click the "Create Chart" button.  Your browser will open, with a
    complete graphic rendering of all chords in your song.  You can use
    your browser's page settings to print charts at different sizes.

    Two versions of each chord are presented:  1 with the root note on the
    6th string, and another with the root note on the 5th string.  Chord
    lists can be saved and reloaded with the "Save" and "Load" buttons.
    The rendered images and the HTML that displays them are all saved to
    the "./chords" folder (a subfolder of wherever this script is run).
    You can create a zip file of all the contents of that folder to play
    your song later, upload it to a web server to share with the world,
    etc.


    -- THEORY --

    Here are the formulas and fingering patterns used to create chords in
    this program:


    6th string notes:                5th string notes:

    0  1  3  5  7  8  10  12        0  2  3  5  7  8  10  12
    E  F  G  A  B  C  D   E        A  B  C  D  E  F  G   A


    The sharp symbol ("#") moves notes UP    one fret
    The flat  symbol ("b") moves notes DOWN  one fret


    Root 6 interval shapes:          Root 5 interval shapes:
    ___________                      ___________
    | | | | 4 |                      | | | | | |
    | 3 6 9 | 7                      | | | | | |
    1 | | | 5 1                      | | | | 1 4
    | | 7 3 | |                      | | 3 6 | |
    | 5 1 4 6 9                      5 1 4 | 9 5
    | | | | | |                      | | | 7 | |
    | | 9 | 7 |                      | | 5 1 3 6


    To create any chord, slide either shape up the fretboard until the
    number "1" is on the correct root note (i.e., for a "G" chord, slide
    the root 6 shape up to the 3rd fret, or the root 5 shape up to the
    10th fret).  Then pick out the required intervals:


    CHORD TYPE:           INTERVALS:           SYMBOLS:

    Power Chord           1    5                5
    Major Triad           1    3    5           none  (just a root noot)
    Minor Triad           1   b3    5           m, min, mi, -
```

```
    Dominant 7            1    3   (5)  b7      7
    Major 7               1    3   (5)   7      maj7, M7, (triangle) 7
    Minor 7               1   b3   (5)  b7      m7, min7, mi7, -7
    Half Diminished 7     1   b3   b5   b7      m7b5, (circle with line) 7
    Diminished 7          1   b3   b5  bb7 (6)  dim7, (circle) 7
    Augmented 7           1    3   #5   b7      7aug, 7(#5), 7(+5)


    Add these intervals to the above 7th chords to create extended chords:

    9 (is same as 2)   11 (is same as 4)   13 (is same as 6)

    Examples:              9          =    1   3  (5)  b7    9
                          min9        =    1  b3  (5)  b7    9
                          13          =    1   3   5   b7   13
                          9(+5)       =    1   3  #5   b7    9
                          maj9(#11)   =    1   3  (5)   7    9  #11


    Here are some more common chord types:

    "sus"       =  change 3 to 4
    "sus2"      =  change 3 to 2
    "add9"      =  1 3 5 9  (same as "add2", there's no 7 in "add" chords)
    "6,  maj6"  =  1 3 5 6
    "m6, min6"  =  1 b3 5 6
    "6/9"       =  1 3 5 6 9
    11          =  1 b7 9 11
    "/"         =  Bassist plays the note after the slash


    NOTE:  When playing complex chords (jazz chords) in a band setting,
    guitarists typically SHOULD NOT PLAY THE ROOT NOTE of the chord
    (the bassist or keyboardist will play it).  In diagrams created by
    this program, unnecessary notes are indicated by light circles, and
    required notes are indicated by dark circles.
}

root6-shapes: [
    "." "major triad, no symbol (just a root note)" [1 3 5 11 55 111]
    "m" "minor triad, min, mi, m, -" [1 b3 5 11 55 111]
    "aug" "augmented triad, aug, #5, +5" [1 3 b6 11 111]
    "dim" "diminished triad, dim, b5, -5" [1 b3 b5 11]
    "5" "power chord, 5" [1 55]
    "sus4" "sus4, sus" [1 4 5 11 55 111]
    "sus2" "sus2, 2" [1 99 5 11]
    "6" "major 6, maj6, ma6, 6" [1 3 5 6 11]
    "m6" "minor 6, min6, mi6, m6" [1 b3 5 6 11]
    "69" "major 6/9, 6/9, add6/9" [1 111 3 13 9]
    "maj7" "major 7, maj7, ma7, M7, (triangle) 7" [1 3 5 7 11 55]
    "7" "dominant 7, 7" [1 3 5 b7 11 55]
    "m7" "minor 7, min7, mi7, m7, -7" [1 b3 5 b7 11 55]
    "m7(b5)" "half diminished, min7(b5), (circle w/ line), m7(-5), -7(b5)"
        [1 b3 b5 b7 11]
    "dim7" "diminished 7, dim7, (circle) 7" [1 b3 b5 6 11]
    "7sus4" "dominant 7 sus4 (7sus4)" [1 4 5 b7 55 11]
    "7sus2" "dominant 7 sus2 (7sus2)" [1 b7 99 5 11]
    "7(b5)" "dominant 7 flat 5, 7(b5), 7(-5)" [1 3 b5 b7 11]
    "7(+5)" "augmented 7, 7(#5), 7(+5)" [1 3 b6 b7 11]
    "7(b9)" "dominant 7 flat 9, 7(b9), 7(-9)" [1 3 5 b7 b9]
    "7(+9)" "dominant 7 sharp 9, 7(#9), 7(+9)" [1 111 3 b77 b33]
    "7(b5b9)" "dominant 7 b5 b9, 7(b5b9), 7(-5-9)" [1 3 b5 b7 b9]
    "7(b5+9)" "dominant 7 b5 #9, 7(b5#9), 7(-5+9)" [1 3 b5 b7 b33]
```

```
    "7(+5b9)" "augmented 7 flat 9, aug7(b9), 7(#5b9)" [1 3 b6 b7 b9]
    "7(+5+9)" "augmented 7 sharp 9, aug7(#9), 7(#5#9)" [1 3 b6 b7 b33]
    "add9" "add9, add2" [1 3 5 999 55 11]
    "madd9" "minor add9, min add9, m add9, m add2" [1 b3 5 999 55 11]
    "maj9" "major 9, maj9, ma9, M9, (triangle) 9" [1 3 5 7 9]
    "maj9(+11)" "major 9 sharp 11, maj9(#11), M9(+11)" [1 3 7 9 b5]
    "9" "dominant 9, 9" [1 3 5 b7 9 55]
    "9sus" "dominant 9 sus4, 9sus4, 9sus" [1 4 5 b7 9 55]
    "9(+11)" "dominant 9 sharp 11, 9(#11), 9(+11)" [1 3 b7 9 b5]
    "m9" "minor 9, min9, mi9, m9, -9" [1 b3 5 b7 9 55]
    "11" "dominant 11, 11" [1 b7 99 44 11]
    "maj13" "major 13, maj13, ma13, M13, (triangle) 13" [1 3 55 7 11 13]
    "13" "dominant 13, 13" [1 3 55 b7 11 13]
    "m13" "minor 13, min13, mi13, m13, -13" [1 b3 55 b7 11 13]
]
root6-map:  [
    1 20x70 11 120x70 111 60x110 3 80x90 33 40x50 b3 80x70 5 100x70
    55 40x110 b5 100x50 7 60x90 b7 60x70 9 120x110 99 80x50 6 60x50
    13 100x110 4 80x110 44 100x30 999 60x150 b77 100x130 b33 120x130
    b9 120x90 b6 100x90 b55 40x90
]
root5-shapes: [
    "." "major triad, no symbol (just a root note)" [1 3 5 11 55]
    "m" "minor triad, min, mi, m, -" [1 b3 5 11 55]
    "aug" "augmented triad, aug, #5, +5" [1 3 b6 11 b66]
    "dim" "diminished triad, dim, b5, -5" [1 b3 b5 11]
    "5" "power chord, 5" [1 55]
    "sus4" "sus4, sus" [1 4 5 11 55]
    "sus2" "sus2, 2" [1 9 5 11 55]
    "6" "major 6, maj6, ma6, 6" [1 3 55 13 11]
    "m6" "minor 6, min6, mi6, m6" [1 b3 55 13 11]
    "69" "major 6/9, 6/9, add6/9" [1 33 6 9 5]
    "maj7" "major 7, maj7, ma7, M7, (triangle) 7" [1 3 5 7 55]
    "7" "dominant 7, 7" [1 3 5 b7 55]
    "m7" "minor 7, min7, mi7, m7, -7" [1 b3 5 b7 55]
    "m7(b5)" "half diminished, min7(b5), (circle w/ line), m7(-5), -7(b5)"
        [1 b3 b5 b7 b55]
    "dim7" "diminished 7, dim7, (circle) 7" [1 b33 b5 6 111]
    "7sus4" "dominant 7 sus4, 7sus4" [1 4 5 b7 55]
    "7sus2" "dominant 7 sus2, 7sus2" [1 9 5 b7 55]
    "7(b5)" "dominant 7 flat 5, 7(b5), 7(-5)" [1 33 b5 b7 111]
    "7(+5)" "augmented 7, 7(#5), 7(+5)" [1 33 b6 b7 111]
    "7(b9)" "dominant 7 flat 9, 7(b9), 7(-9)" [1 33 5 b7 b9]
    "7(+9)" "dominant 7 sharp 9, 7(#9), 7(+9)" [1 33 b7 b3]
    "7(b5b9)" "dominant 7 b5 b9, 7(b5b9), 7(-5-9)" [1 33 b5 b7 b9]
    "7(b5+9)" "dominant 7 b5 #9, 7(b5#9), 7(-5+9)" [1 33 b5 b7 b3]
    "7(+5b9)" "augmented 7 flat 9, aug7(b9), 7(#5b9)" [1 33 b6 b7 b9]
    "7(+5+9)" "augmented 7 sharp 9, aug7(#9), 7(#5#9)" [1 33 b7 b3 b6]
    "add9" "major add9, add9, add2" [1 3 5 99 55]
    "madd9" "minor add9, min add9, m add9, m add2" [1 b3 5 99 55]
    "maj9" "major 7, maj9, ma9, M9, (triangle) 9" [1 33 5 7 9]
    "maj9(+11)" "major 9 sharp 11, maj9(#11), M9(+11)" [1 33 b5 7 9]
    "9" "dominant 9, 9" [1 33 5 b7 9]
    "9sus" "dominant 9 sus4, 9sus4, 9sus" [1 44 5 b7 9]
    "9(+11)" "dominant 9 sharp 11, 9(#11), 9(+11)" [1 33 b5 b7 9]
    "m9" "minor 9, min9, mi9, m9, -9" [1 b33 5 b7 9]
    "11" "dominant 11, 11" [1 b7 9 44 444]
    "maj13" "major 13, maj13, ma13, M13, (triangle) 13" [1 3 55 7 13]
    "13" "dominant 13, 13" [1 3 55 b7 13]
    "m13" "minor 13, min13, mi13, m13, -13" [1 b3 55 b7 13]
]
root5-map:  [
    1 40x70 11 80x110 111 100x30 3 100x110 33 60x50 b33 60x30 5 120x70
```

```
    55 60x110 b5 120x50 7 80x90 b7 80x70 9 100x70 6 80x50 13 120x110
    4 100x130 44 60x70 444 120x30 99 80x150 b3 100x90 b9 100x50 b6 120x90
    b66 60x130 b55 60x90
]
root6-notes:  [
    "e" {12} "f" {1} "f#" {2} "gb" {2} "g" {3} "g#" {4} "ab" {4}
    "a" {5} "a#" {6} "bb" {6} "b" {7} "c" {8} "c#" {9} "db" {9} "d" {10}
    "d#" {11} "eb" {11}
]
root5-notes: [
    "a" {12} "a#" {1} "bb" {1} "b" {2} "c" {3} "c#" {4} "db" {4}
    "d" {5} "d#" {6} "eb" {6} "e" {7} "f" {8} "f#" {9} "gb" {9} "g" {10}
    "g#" {11} "ab" {11}
]

f: copy []
for n 20 160 20 [append f reduce ['line (as-pair 20 n) (as-pair 120 n)]]
for n 20 120 20 [append f reduce ['line (as-pair n 20) (as-pair n 160)]]
fretboard: to-image layout/tight [box white 150x180 effect [draw f]]
; spacer: to-image layout/tight [box white 20x20]

view center-face layout [
    across
    t1: text-list 60x270 data [
        "E" "F" "F#" "Gb" "G" "G#" "Ab" "A" "A#" "Bb" "B" "C" "C#" "Db"
        "D" "D#" "Eb"
    ]
    t2: text-list 330x270 data extract/index root6-shapes 3 2 [
        either empty? a/text [
            a/text: rejoin [
                copy t1/picked " "
                pick root6-shapes ((index? find root6-shapes value) - 1)
            ]
        ] [
            a/text: rejoin [
                a/text newline copy t1/picked " "
                pick root6-shapes ((index? find root6-shapes value) - 1)
            ]
        ]
        show a
    ]
    return
    a: area
    return
    btn "Create Chart" [if error? try [
        make-dir %chords
        delete/any %chords/*.*
        ; save/bmp %./chords/spacer.bmp spacer
        html: copy "<html><body bgcolor=#ffffffff>"
        foreach [root spacer1 spacer2 type] (parse/all form a/text " ") [
            diagram: copy [image fretboard]
            diagram2: copy [image fretboard]
            root1: copy root
            foreach itv1 (third find root6-shapes type) [
                either find [1 55] itv1 [
                    append diagram reduce [
                        'fill-pen white 'circle (select root6-map itv1) 5
                    ]
                ] [
                    append diagram reduce [
                        'fill-pen black 'circle (select root6-map itv1) 5
                    ]
                ]
```

```
            ]
            append diagram reduce ['text (trim/all join root1 type) 20x0]
            append diagram reduce [
                'text
                trim/all to-string (
                    select root6-notes trim/all to-string root1
                )
                130x65
            ]
            save/png
                to-file trim/all rejoin [
                    %./chords/ (replace/all root1 {#} {sharp}) type ".png"
                ]
                to-image layout/tight [
                box white 150x180 effect [draw diagram]
            ]
            append html rejoin [
                {<img src="./}
                trim/all rejoin [
                    replace/all copy root1 {#} {sharp} type ".png"
                ]
                {">}
            ]

            foreach itvl (third find root5-shapes type) [
                either find [1] itvl [
                    append diagram2 reduce [
                        'fill-pen white 'circle (select root5-map itvl) 5
                    ]
                ] [
                    append diagram2 reduce [
                        'fill-pen black 'circle (select root5-map itvl) 5
                    ]
                ]
            ]
            append diagram2 reduce ['text (trim/all join root type) 20x0]
            append diagram2 reduce [
                'text
                trim/all to-string (
                    select root5-notes trim/all to-string root
                )
                130x65
            ]
            save/png
                to-file trim/all rejoin [
                    %./chords/ (replace/all root {#} {sharp})
                    type "5th.png"
                ]
                to-image layout/tight [
                box white 150x180 effect [draw diagram2]
            ]
            append html rejoin [
                {<img src="./} (trim/all rejoin [
                    replace/all root {#} {sharp} type "5th.png"
                ]) {">}
                ; {<img src="./spacer.bmp">}
            ]
        ]
        append html [</body></html>]
        write %./chords/chords.html trim/auto html
        browse %./chords/chords.html
    ] [alert "Error - please remove improper chord labels."]]
    btn "Save" [
```

```
        savefile: to-file request-file/file/save %/c/mysong.txt
        if exists? savefile [
            alert "Please choose a file name that does not already exist."
            return
        ]
        if error? try [save savefile a/text] [alert "File not saved"]
    ]
    btn "Load" [
        if error? try [
            a/text: load to-file request-file/file %/c/mysong.txt
            show a
        ] []
    ]
    btn "Create Zip" [
        if not exists? %chords/ [alert "Create A Chart First" return]
        ; rebzip by Vincent Ecuyer:
        do to-string to-binary decompress 64#{
        eJztW+uP20hy/+6/oldGsJ7bcEU2X00Zd4bX9iILXO4AY5N8EOYAjkTNMNaQOomy
        PTbmf8+vqptkU3xIM4cgARICHmvYVdX1fnRrPn745a9/FsvrFy9W1VfnW75biFVZ
        VNnXSixfCDyr/crZlsXtwvzeeVz885IkSsJEJYEQju96bhCHrhKOF8s4CGToSgKS
        QeQHnh/jo1KRG8aRFzb0HD9O/CCMPA+fvciPwziUEX4RMkhkpEAIH90gAGFf0j6h
        lCqUbhTRPkEcY3dftvx5kUwCMBYSCU+GIO1KIh64sR+5oAo8FUWhBDnwJIJEhX4U
        Bgq4IJbIOIqilj/phknoRrSMz1GcuJGrSC5wBCRFvHoKLISQzSchVOwr0PMJHjyr
        JArjlj8CVH6SsP4iBQFiLyR6TuCrIJRhABJJ7LsBRCQg3w8j3w8U6S+KPGI1avUn
        HNggCsKA9ZdIpUJgJ1hIkhgSu0lEJlBeEvpgFkDYzvO8JKJ9wtiVkCgMGnqe60ZK
        KS8I2XQkVBzR3k4cuxArimLCk6RJ6I32gd2TOJaQgei6RKC150Bb0JRL8jqeD39x
        vTCCMgWwYdYgJP3BYSRkc0mIEDYIvdCnfQJP+lC+77XyRqEXwwqK9efHWPahENID
        VOdJ2A94xHMko4T2CaMwdGNJvEKqIJJebNFzlAv7h2HC+oOWwFWSkFxR7IGnhPzP
        C8mllMv+l4BXP/LZ/+BCkZcElrwiTkLlh6Ek1/IC2N93pQyJJwgC5cAXHUgVKC8G
        HxxDgUdUJHuH54VQTEPPIWbJA2lrOBR4ChLaG3huHICviA0FVv1YerRPhF29wA3I
        F32yrSv91v+kC6XFiBGXfwtJ867PsYQHvkN2dPA/wGQYUQz5McweRz4LRL8pz2/5
        g/IilYRxzC4sOSEkCcUERE1CWJhihewRKNIZ4ilCRMBTSR4J51CBp1r/U1BO7MKY
        gmMM1pCkJ0d6HqJAQgO8qx968EyyaUwBqRLONT4JLmPX0h+CN4CTUIoiwYIkSOKI
        bBrBt0PF9AQ0CgvAiwnBlyQF0Yb34iUiuPUXz4e/+WHCoSkSFSSkb9JfGMR4i1RF
        sSIhYBKHimMoAo8wNwUXtAd2PK/lLyQ4KJi3iwOIgqDlmIDh4bBuRLEC0jHUyj6n
        wlj6yqd9EBmsY7flDyaCjwQR2QMxBu1CZxxLgfQofkOOMWDCIsQTlBr6fkIJDsGF
        gEx87Nky6LnIXyCpONmQTZFWKdgD5B0w5VKyAZ4HHn1yQGTDIPDBIUUXIgRJTXmi
        5RCxhtiCY3K4wEt8BY/kaFIhJQ+OZuI2Voqjzw0SlA6XRIopuCjFWhyCHx+lIuTs
        66K6yMAnYJQkpCvgUkpAMqCADtlukfTJb8g8cF3lIUdJYdUkbAG2Ql2TAs6Yih0S
        kQXnTEhvKDsKviBp10C5LmKdc4JykbvAsUVQIANQRfRdjkCsJ+QJhIjcmniUvYEo
        FZiXigMNsuMfBbFEdkXSQ9RaHEqsgRUv4qqEihdHiUfUkQ6hfeQ7iowIe8HzyFNg
        fqRBN+RIgpeBhcAyigigGiklhx1kRe2OFZcRBEQEhmOXPikZIf0Q50IidFA3JaWi
        GC4ETwttHcLACAJPG4WqgkpYocJHeUKyIYdHrZYI5Yh6A4QatQYB9w+hQlFBeCuL
        QxSwGLHic0/hB1iOeHeH0lmAHEE6pOB2kd042FDIoE1FwQS3jVD/wsTiMIopMwfs
        bjC2knBu9nKUeKgAlYKIJFAg0iRnxQAMIF592gh5TmFjyygCFHyJxKkzMrIkJSny
        IfBBbQqXJo96IRmw90ewODZyyQ996BaeEUUWh/BYdFcel1+PyiYeyeEWI4uiDHFN
        g02gXS/k2gRXcWPuIZDEAAAci8PYRz1CeQ04DkPqtJKQnNbF5hQSFL+UPShwyFho
        xqjIUGNDXurD16SVvOC9iuwScdQgUSJsdXFC4kQajbmoUeeFxEuGjdDt+DCdx1GG
        foBSXJsaoDwXJLmYQJ9YhqOSXWNoB4mUTOJQQGMXlxtDMIi2hPIRVSrIg+xnpQbl
        I/BhK86uCFgoyKVWE0EVkOZZSPg1jImmh2IJFoQpKO84ktw+QuWykleM9EgFjgVD
        oUYUwru5PCnkyDgMOakjn8Nu3BrCi9F3UkoXAedMFNOWQ9RjlCBASM0wQiDi2ujF
        HrWHIduKOl4F0Ti/oByhpYkSTn2o2S6ylcUhmm1k/5gbYAHTkBlc0mGETgcGcqkT
        g1DcuLtc0UARfazkIKPONbECz6HyFlAAkQQOJXzEBHsekij6IF2t0PpT4eUCAGFg
        4iCmCg/fRGsCh7I6TPS/KI+xrqA+Eheyp8spCnUT/knujnQCD+Yy5lGCg4dziUQd
        hVfDjnaFgmIQooorFNIMFVD2Qm4GYBKSDGWBLBLSRtQ6QHpu19F9INTdxLdUGEgq
        skFAoSYirsKBz30bumvEleIeE70cQjIgFYZU0PyYUomA3WDAGA2bVaGgnASxy85B
        WQfr3PyFQEMB5pQBRqkwetysYSoj1H1+e1c5h7t8UzlqITbHYmUNY7OP7fJBVHeZ
        +Jxuj5m4eRBK3OR4lxZrsc+q4744iLz6edagasBljnnvNtv/cC1mvzfoVSmYpIa+
        bvfLcuyxF0V2m1b55+xNTaUzHTqe+FruzRLt71AfORf0kxawTm8pFVDH2uBen5Ah
        eALUhBi/hX3R/uQfx906rTIHE+uAkoz8pB9ApTTXWprgV8u0eHAO1T4vbkkX7+kd
        1LC6y1afDsf7Fhq4Ha39VuRVnm41lz2Fbcp9lq7uQCfdm426kzQz/KpjZXp3xZra
        5atPzRDOO5M+UGp5lWn+JLyeUprZ3ZfjykjFu4/vUEtrCZ+lkL5rZPe76uGNIfHy
        u2uexxPjbst0Lf6zzAsxe/l9JpZV6dxlX2vXaY2pCeH17HF2PSzo60N+W6QQKju8
```

0KRX6dbZ5NvMwcoCTBsxxMvvCONfXEydj1pFWVHtp2ERqxo2K9ZOuXFqlGFoNK8a
mrh21tlhtc93VbkfAcdQ8GhkWJXF52x/yMuCDaYlAUKuj19OjfiOwTm8hfFFMk4q
tnlVQRhwm2OJkCdD/t/rcNf79wMeLNzkRbp/QBJhc3XNiGX2wjbW4ZyDECSLYfRV
C42WU7lXJ7F9Ho2zZKQxUQn9S3HnddKJrgYzCWEc7sp99WyNM/b/PpWfS6EAfaKj
3eS3/7CXrcuBFKDRKdw1a7dZdYFRhszAjDZs9zm0EpzYAX6KVUuXd1lKJZXCNbOM
tSp3D/Ndim01eWmxf063A2H7P8t80LjF/WFdHpwqv88mBGCPYEhBkH1u6e2lzq+N
bUJ2flce9+IPdACsrjpeXtZRPb/Pi2OVAciXfZDTIJgfMuy9pli46gaAlpQKz2WS
EmRfUnr7RElptvyDeHWffhWuYfIhQ/w6NJe4o1KXRXWnhRZ1tzVfpw+WUOR7F9lv
sy/vz1jwKf7WxV1YIazf2Dqg3eCQ6+PqtI3E0ICGvtMAwgk6IJidui2ifQ8g6DTQ
WoUTDea+VktnbN/R0rD1/7u0RLuNaCmkQ2lbB+ROP7HrdAAD5V6qqvEu+1u+c6gD
emjU9P1deb9D63XIKI9RJ9UZOSxmuS/TEJSFMupdVzXymhdaYNNnafB1vs9WaKMe
BO+toa4fG+gihQ8tCZT0/Rf6rdwwaqentaLzX8t1vslXGGLQcPHSEMJ4E2zY1tBz
lqwzINxn1V25tsRzmBxU1r455N8ycfL7qWu8RgHYlEIziaKb0TzRIIF5e9dFN7vr
AYARG6geA4umyRTbrLit7nTr/mKyre84jhZ1IX48wEKW63ShXotv2/xGQJVZet9Z
OdHRonnR5dxoY2MLD+VkG5gxh1v8s+D92xkvLw4V1NEhYIR5ZYt6Jf7Uvjhh5mrg
3rNm0hTM3jo9h0/5rmd8OQ5apfm2B++oHkKj63W22cJrOwDXfV5HbNOofktl5mTf
Lk0rFYw5Zbqpsv2wTz7R1wYT3Gs7cVjxbx5uIF9+fxywVHcQ7C3r4fQRG5jJawIC
2r499NaNO5lg/6NR89BdeYdYAz9slQE7Gnx1gj/kBUyh96bTyHHCm9OncTjOiQy3
HtqBchw9r02W6QMM5DX76SWiIVZMNaydhdP8a3YE/qjfT1gt+4oassmz7XoMlumw
HDXVQd1rkKK0KA7mBaOStqhNBJSBNecJKHHP8PDTA4zzPi4O6KdXfSn7gOiq/z8g
/s8FhGVhUAVL93Cx88Dr/PCpON7fwPiHaqgqtqA0je0LJPS0Q191g8mt70fWuaHm
+okYpr8sN5tD1uflSUE/EfEtkNHTaOnUP/k/aihp2zfjJ7O/f/y3D3Rw3HbTx705
Xz78PLNHjephh3bWNI14oZvg+lQeWObjtdkcLrRLqztmoLf/92bQAeIP1OZyR23a
+fvyMzp8UiwdYM7ms7b9NgxpaMK9ZOjJdb89fIOAxZfYAhG9yfeH+lBiaQYl6r1E
Qd9S08J1UPXgMTjHNLwukChW5TpzGr0265xs/2k2O2GoPosAL+Crs8Rq1+ukWzGb
XY+tIzIw5HUgOr4BZxU/YsRqZqtfjvl2TXMVvUz3q7v8c1aPoewccBJMBWh0PtUD
zOFnTbt2JhOVWKVakcOGENCQ+rm14Rfk7Y4NhT4Z+kG0089/MAxsekNsibxqFaGr
ShefuCK0X4krQsuL1fa4ziwGWgLzdZbt6DKFQQ5aFII8HG/q8Q/MWwjwqpsSY87s
z/mhqhG+3JFWml60uB2c0DgBcL3maiuKG6dWjiZTIOvDP9o0Y5VpM4kyoJ7hymMl
mqJguTje10GmHbWuj00OqNXecZglJhdEjFk7cfGTWrhkoIWYRLFjrre1IVDusmKu
pZtry8+/7HOUPF63mv9W7oWlt4Vwx3VlIhYtzIsTh9HDHllx+Uq/ubq2XJJsuSzK
iuejNwbnRANky4VZmnvdJWPFhY5BTofa9Plef+pGKotmgHvJPt1uxbJBNNvid3IA
frOn2ZuJLrtkT1u+/KARUkGi8WDdnnC8GZpX9d4UIA3GG83t9QCnr0W6gzXX+ru8
RVVncrFFoAwWsfqSkqEaOUbaNnqMu/HYakKfMOaL7iHOoAIGujnbj6xYtK8260cf
0RXllxcDcpNmqSyUhQlvU0OJtzdDpDDItyEpltrNmISRqg0X42FypBk1ZEzV2qYo
FFqHteuP4LE2N2zX2l0n9N5qgA8tGZq3+fHeOtByBjvU+un3wvVDljfBL0a7oj76
iGS6ctdxbin1gu5cR5fdrQwzBM3VZWC5Q5mqBqKantekr9PDRvjX4MGi9ZijzubU
s6kcWX+y0/vopKlrM3efVrXrzwR3aXGb6SMgpj+XIpDtKUmbawd2Ose7HaP98mW2
G6Db15LupnugVPY0mX6Y2lXCqpQ/NaPLEOJpomSfQcvUeT3lUC0B4m1kbu7poptx
e9f8neXuxEPN1dRyfYiNrTpg7SRnZTvIe9P0aIT+NJQedOecbVrocw7XisXNqWmx
piZDevrQwy0J3ZM07chqS8Fsmo4aqBW51y0fC7tffp+t7NsIu2vmi7vW/9vG2fTL
fy22D+JLuf+EJhK5vHPMrM8XDvVxBg1E9Qms1ULTo6Xo9MFUQupe2Oqh1w2rnUaa
Bes3091riLetTC2ZLo2p9rjFoVHuFSRJj9vq6oTA3485zSrvy+LHiluHCSKDXTYf
EQ3chHAmGD/uoBTraLfS6badwOvXXNubg5sGs0OTYaxY4dmMnZEPr7P9vtwfTjt2
/dZuZgl4pImnpkwriaq30fdpf16gFxnv3qlmma9AzJ7VtpsG9CReCIG4O8Xi7vEr
mdK8OeHnPv2UUYrQwxhDnGOn9tU6Lds9hOnXB8KYNNy83qX7QzYnhmtiJ8lp6iD/
UA42qtXd/nju/P/VeNt51QOWukqP3xVwn6ddXsMO5sRXptf7lu3LNxp8br4NKZbm
6GKTbuFHfRZ4BxNOU1vUlz7WfbJ+NUKSA8nI90p/UaD7xQGGGEHmXqhGHuTH6le7
B6mj0HPNxOB5bEuR6Tg0Csy/Ich6kH2Gg9qGo0e0u4deogomNN2732q+7HNKZkR7
/fvgqe0GLm/bDXuLI1va6XXKiyy4jitZ70d26OfrqX160J3deqsTUnVEm3DIZrgY
nyvo4TIxuNrnoS/yoMDd++3a5IOgw6wf0Jqs7uamYtfJYHxedM/Mkr2KuZgEp6f9
ppj+JvWZSwv7YY3O/uYI50+i/ISaQd3U9d9OzlPtZ3xWVeen5Ha419m2H2zTJGpp
+eBqCmhitK+fM9/pHHteflf014sfgviRmvD37lvvrfuWGvHmO88X0jGXI+8J97d/
ef9xqnE/fWaEMAMiXTcsxsbBoQfzcL5GNPyxo3xx1lPqx3wt9YLLsaGnkdrTd0L0
HXuw4qGbHagXIxQU4d7kcPdsd6G+aN9HnqS3GDNIadj1dp8+HNCPXM76ozWMt1PI
k9CR4Sr7PrYYKpRTBIpS39dt04FzmzHE9h6uaP9e4EJ/eYJb1s5B33aLOt8lu9xF
Zr+9f/t769qDhzpDz8vvsUrePdZfa3pSSLhsiNo2dM8Hvyie4pOuoD+OFL9++PVX
AUVfhPYP5G175+fbtov9pPTz4S/vWxtlxeUx8Dxux6uOXp1crr8vR5PkG0Fnhktd
i/gvXujjBeXCrpSbFG3Kms7/+cBhsl42TJg5Vv8/eJB++vRbgfPZ4gJJntFh0EOT
aHMD+fTsv892WVqJ/Fk1nx778LQnwkUU6Kn/EIP/jIt/6G8mXmCOaR/ULHa7qUv9
Y5zy8MrUxQWxsDzxU1NsZnXVmRFH45s+wVWf4qLDG1r3r3wkN3wBaz9D96uT3xgZ
QrBPjCZtc3p9NJ1p9D3g5e55rU+jlj2MJxv+tQ7Q9jg15b+oQClr7z/GLT52VTY9
PGz6x1fzxQWI9AwcbGnc50bK9JZ0Mr1sDhyvxWG3zSu+yzpv1WE5GfeZchLuM+U0

```
        DPUPVs+zwo5QnwReqvH6eXrKPTT3oU/zDHp6d6eL8fMp+znXJzxtpf+2e87QXa/K
        TiNknc6anHxSBfS3cObvc+TYY2EuRdYLpGnr5LOP1V6o0ARS5+pZna07CC0Pet62
        j9hfoGv6L0cgfyluTAAA
    }
    zipfile: to-file request-file/file/save %/c/mysong.zip
    if exists? zipfile [
        alert "Please choose a file name that does not already exist."
        return
    ]
    zip/deep zipfile %chords/
    ]
    btn "Help" [editor help]
]
```

10.17 Case 17 - Web Site Content Management System (CMS), Sitebuilder.cgi

For many years I used a simple CGI web site manager called Chico WebTool. Together with a javascript WYSIWYG HTML editor, the WebTool script enabled an extremely simple way for users to add, edit, and manage page content on web sites, without installing any software on a client PC (i.e., no Dreamweaver, Frontpage, etc., needed). New pages could be added directly in a browser, from any computer, and they were automatically linked in a sub-page tree structure on the web site. Users selected from a list of HTML templates (which were very simple for designers to create), to give the entire site a consistent look and feel. By adding the third party javascript editor, page content could be edited easily by users, even without any knowledge of HTML. WebTool ran in web servers on any operating system, as long as they had CGI and PERL installed. WebTool created static HTML pages, and didn't require any SQL database. It was a versatile setup that worked absolutely intuitively for novice and experienced users alike, long before any of the modern CMS packages became popular.

One problem with WebTool was that it took a long time to install and configure on each web server. There were also a number of features that I wanted to add to it, and some significant changes that I wanted to make to the workflow. The web site where Web Tool was distributed is no longer hosted by its author. So I decided to create a similar system in REBOL, more exactly suited to my needs. Here were my thoughts to get it going:

1. Start with a simple HTML textarea editor to create and edit pages. The password protected editor presented earlier in this tutorial is a good start.
2. Create an interface to add new pages, and to select existing pages to edit and delete. All that's needed is a text field to enter new page names, and a list of links to existing pages. The contents of any existing page would be read and sent to the textarea editor, and the edited text would be saved back to the same file after being submitted by the user. If a new file name is entered, the new file should simply be created with an empty string, then sent to the editor.
3. Integrate the code for a javascript WYSIWYG editor, to automatically enable visual creation/editing of pages in the above editor. I decided to use the openwysiwyg editor from http://openwebware.com because it is stable, small, runs in just about every browser, and enables many essential features. The openwysiwyg installation is composed of numerous files in several folders, so I decided to compress it with Carl Sassenrath's rip archiver. This would allow me to embed and extract the whole package on any operating system, directly from the web editor script.
4. Create an interface to upload images and other files used on the web site. Andreas Bolka's decode-multipart-form-data function, explained earlier, will work great to handle file uploads.
5. Add an interface to run OS and REBOL console commands, to manage files and folders, to create backups, to download files from other FTP servers, etc. An entire script that does all this was already presented earlier in this text.
6. Create a template system to wrap all user created content pages in a consistent overall page design. User created content will simply be inserted into a table area in any template's HTML layout. This gives the entire site a uniform look and feel, without any work or general design by users. All users have to do is type some text, add some images and/or other basic content, submit it, and the page generated by this script will look complete. To link pages together and create a navigation structure on the site, the template system should create several menu areas on each page, with automatically generated links to other pages on the site. I wanted users to be able to add new pages as *sub-pages* of any other existing page on the site. The home page should be able to have links to as many sub-pages as desired, and each of those pages should be able to have as many sub-pages as desired, and so on, for as many levels deep as desired, to create a simple tree structure, with the home page as a starting point. The system should automatically choose between 2 basic template designs - 1 with a link menu area (for pages that have sub-pages), and another without any link menu area (for pages without sub-pages). I also wanted each page to contain a separate menu area with links back *up* through the currently traversed sub-page tree structure, on every page. This would make the entire site easily navigable both down through sub-pages, and all the way back up to the home page. This organizes every area of the site into clearly divided sub-sections, and enables visitors to instantly know where they are, and how to move up and down through the tree structure. The site map and menu links should be built automatically by the script, but should be manually editable, directly within the script, using a simple syntax. Template layout pages should be easily editable by a web designer, or even by users who know basic HTML, to change the entire look of the site, and to easily alter static elements found on every page (logo graphics, color

layouts, copyright info, etc.).

Steps 1, 4, and 5 were already covered in scripts described earlier in this tutorial. I would just need to incorporate them into a new script with steps 2, 3 and 6 above. I started out with this large amount of code that I've already written for other situations, and covered earlier in this text:

```
#!./rebol276 -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Sitebuilder"</TITLE></HEAD><BODY>]

; Read the submitted GET or POST data (standard code covered earlier):

read-cgi: func [/local data buffer][
    switch system/options/cgi/request-method [
        "POST" [
            data: make string! 1020
            buffer: make string! 16380
            while [positive? read-io system/ports/input buffer 16380][
                append data buffer
                clear buffer
            ]
        ]
        "GET" [data: system/options/cgi/query-string]
    ]
    data
]

submitted: decode-cgi submitted-bin: read-cgi

; If no data has been submitted, request user/pass (as demonstrated in
; several earlier scripts):

if ((submitted/2 = none) or (submitted/4 = none)) [
    print [<strong>"W A R N I N G  -  "]
    print ["Private Server, Login Required:"</strong><BR><BR>]
    print [<FORM METHOD="post" ACTION="./sitebuilder.cgi">]
    print [" Username: " <input type=text size="50" name="name"><BR><BR>]
    print [" Password: " <input type=text size="50" name="pass"><BR><BR>]
    print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="submit">]
    print [</FORM>]
    print {</BODY></HTML>} quit
]

; Check user/pass, end program if incorrect:

username: submitted/2 password: submitted/4
either ((username = "username") and (password = "password")) [
    ; if user/pass is ok, go on
][
    print "Incorrect Username/Password."
    print {</BODY></HTML>} quit
]

; Here is Andreas Bolka's decode-multipart-form-data function,
; wrapped in some standard CGI code (all covered earlier in this
; text).  It's given here in compressed form, and written to the
; server to avoid some of the issues that can occur when
; deciphering different types of post data.  Uploaded binary data
; is simply sent to a totally separate script (created below), and
; then returned afterwards to this script:
```

```
if not exists? %upload.cgi [
    write/binary/allow %upload.cgi to-binary decompress 64#{
eJyFV21v2zYQ/m7A/+GqoYUDTFHaAV2h2A7aNVsHpEixZhgGwxsoibbZSqJKUs2M
LP99d3yRZNnFBFgv5N3xubvnjvR3T84TxTNZvvjxJcS5nk5+u35zewOr9XTSKFEb
eMhlbXhtYrNveAqG/2OSnanKv5LHILKav7t7f7Ocv7t+/XY5v/v17uZ6GX0Uhmet
KAuuonniBueJE3lz+/bPJa4wnSjOijjfihQ2bZ3DKillzkoomGGQtZsNV+vVdAJ4
6Xth8h3ovTa8SmRjhKx1gqqI/0vLtYkrbnayAC9PV/Th9uNdNByhi4ynULHPHLRB
B7ZP4PnFi4tDIbf4WOzlD69Gcvc7UXJYNVILI77yK7AeCRmANlIZnYi6aY236ays
R6DoYk3D62Lo/LFMXnKmTs6u+8/Ba/TLNUXA+XwieBg6tY+dg17NP0hlOrFZKngu
Cx5XbWlEw5SJN1JVsbPp8uZUmnhIljCGsTGxs0YDPsWl0AZyDAo6zEtRxRnf+jcK
Qi3rOFf2UW78F75UouIxQUBkYVWylEIum72lLQ0JUjGwkW1dXMFGoMFDbBB1ziSd
MxgnTJ9pVe3QWd9t0IP9Uw5mRQqfpKghiuPIidGCiWGiJAejjFAwtV9EPrDBXa+H
ESDfhpMYAT9pvc6KgMQFInVUkFVT8grRQL5jSnOD6Is2RzqiWkiji+D/a9T8vhQ1
H6rR2qS4Ctn4F/qUeLkuISg4oPQMPY8LoRtJkbMhS0mT07jrJJFtJU3JRB3BWa/a
s+Gyt44lSHc7h41lQP0oZMSuRlUoazTuEhFAAKv3PYn6aNOF1YAeRodtrte3bDlS
H5aY/b4Mbp7C6Gy5+X7UyAHfvZHeYdsJjk3NYMfqosRi7CQ7JwPagOSsq2f3MtY8
LF5fwN7YaKyn+0Gd90O+qk3VQM0qDl9Z2dp73HgUp9dJoSEiDkbg4XIRPQbMloto
EHceGSNh43upCphpXnKsrYFWRFLRgEdcmB122xmx7oqAYf871tpg+x5r+gujBbNR
CwndY8Dds/H+Yl1Pj/O9/oag3WJk9gmhPRmL0BUg+h6EnhzLONaGHjXOV7i6SvSx
caHpsukKnRPQUbsb0t26crDXHMXcpj11QbOdlOAfWlhRB6q3PJF1ibWF4XRatkdJ
TR1pthGKdLu5M5jZkJ0hwvXYnN883bbS2xiyhlCcsNF50FqVb5bTIYOJtQkry+EG
h9paIvn72nR7wqBhdIsRUL+9dvophPPQdIK32JHCUgl36Dan5vKtnfjU5n6w47mM
9nvxdHIJjZIZznRLYQci30u212B9wyU0yA1gWkG3GTYjwwvolve0JVN/KGxUlqvU
2DRXX5EIrcZjhVWWSmxFjR0isPl7W1+4S4vN3kpg9FVKsPAjJqkUSobJRDjCYDTN
jlqAXcD10w510uykkUmwPJ3cE5Ykw/WUte21xgpdhfqT7jzHL66WcwY7xTeL6DzR
/Sn2HPWvyP6CkNLLs4ZpvaAb0euZi8/CPaLlG5Z/plAMDsLzhC3nmVo+9ksaliG0
e1GY3eLVxVPI0BJXi+cHMgjJFF7o+cXFU2skoO0FHbHmSBVZb5e/f7i5ff0Wfrp9
/+Hm+u56nvhxq4u/Q+mfMUIa8lYptIoViUcPsxMaj1AEPD2hjev6SSS9Mpa68PSc
RvEVj+o26E6kOyC6/wuK27PN6mEQ6Ucn/xAB7vRbbhZ/ZyWrPy/9eBRiF63XtmxC
cAhPYgq62TeKJz7tXwz8v0F/TBDpl1aY/wDPpuhm7AwAAA==
} [read write execute read write execute read write execute]

    ; I added a little error check to be sure permissions are set
    ; correctly for the upload script:

    if error? try [call {chmod 755 ./upload.cgi}] [
        print {
            <center><table border="1" width=80% cellpadding="10"><tr><td>
            <strong>./upload.cgi</strong> has been created, but there was
            apparently a problem setting permissions for it.  Please be
            sure that upload.cgi is chmod to 755.<br><br><center>
            <a href="./sitebuilder.cgi?name=username&pass=password&submit
            =submit">Continue</a></center></td></tr></table></center>
            </BODY></HTML>
        } quit
    ]
]

; If a username and password have been submitted to the script, but no
; other data, print the main start page:

if submitted/6 = "submit" [

    ; Print the current working path (by default, where this script is
    ; installed) - just to let the user know what folder they are working
    ; in on the web server:

    print rejoin [
        "<center>Path: " what-dir
        {<br><table border="1" width=80% cellpadding="10"><tr><td>}
    ]

    ; Here's the form to upload data to the upload.cgi script:
```

```
    print rejoin [
        {<br>
        <FORM ACTION="./upload.cgi" METHOD="post"
        ENCTYPE="multipart/form-data">
        Upload File: <INPUT TYPE="file" size="50" NAME="photo">
        <INPUT TYPE="submit" NAME="Submit" VALUE="Upload">

        ; I added some new code here so that users could click a
        ; link to see the existing files on the server.  This link
        ; sends some GET data back to the script.  IMPORTANT:
        ; *** The data after the question mark in the URL appears
        ; just as if it was submitted by an HTML form ***.  When the
        ; script sees this submitted data, it runs the appropriate
        ; code in the "listfiles" section below (see the section with
        ; '"if submitted/6 = listfiles"'). This technique is used
        ; throughout this script to run "subroutines" in the CGI code,
        ; to perform various actions.  This allows us to use 1 single
        ; CGI script, instead of many separate files (similar to the
        ; webserver management script described earlier in this text).

        <a href="./sitebuilder.cgi?name=username&pass=password&
        subroutine=listfiles">Files</a>
        </FORM>

        ; Here's the form that sends data alerting the script to create
        ; a new file name.  When submitted, the script will run the
        ; "edit" subroutine using the file name entered below:

        <FORM method="post" ACTION="./sitebuilder.cgi">
        <INPUT TYPE=hidden NAME=username VALUE="} submitted/2 {">
        <INPUT TYPE=hidden NAME=password VALUE="} submitted/4 {">
        <INPUT TYPE=hidden NAME=subroutine VALUE="edit">
        Create New Page:
        <INPUT TYPE=text size="50" name="file" value="">
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM>}
    ]

    ; This link runs the "console" subroutine:

    print {<a href="./sitebuilder.cgi?name=username&pass=password&
        subroutine=console">Console</a>       }

]

; If a constructed edit link has been submitted, run this edit
; subroutine.  This code was presented earlier in the tutorial:

if submitted/6 = "edit" [

    ; Create new file if it doesn't exist:

    write/append to-file rejoin [what-dir submitted/8] ""

    ; Backup (before changes are made):

    cur-time: to-string replace/all to-string now/time ":" "-"
    document_text: read to-file rejoin [what-dir submitted/8]
    make-dir %edit_history
    write to-file rejoin [
        what-dir "edit_history/"
        to-string (second split-path to-file submitted/8)
```

```
        "--" now/date "_" cur-time ".txt"
    ] document_text

    ; Print the HTML textarea, in which the text can be edited by the
    ; user.  This data is submitted back to this script, and when the
    ; script sees the "save" value (submitted/6), it runs the "save"
    ; subroutine using the submitted text data.  Closing textarea
    ; tags are replaced in the editable text, so that they don't break
    ; the actual textarea in which they are being displayed:

    prin rejoin [
        {<center><strong>Be sure to SUBMIT when done:</strong><BR><BR>
        <FORM method="post" ACTION="./sitebuilder.cgi">
        <INPUT TYPE=hidden NAME=username VALUE="} submitted/2 {">
        <INPUT TYPE=hidden NAME=password VALUE="} submitted/4 {">
        <INPUT TYPE=hidden NAME=subroutine VALUE="save">
        <INPUT TYPE=hidden NAME=path VALUE="} submitted/8 {">
        <textarea id="textarea1" name="test1" cols="100" rows="15"
          name="contents">}
        replace/all document_text "</textarea>" "<\/textarea>"
        {</textarea>
        <a href="./sitebuilder.cgi?name=username&pass=password&
            subroutine=listfiles-popup" target=_blank>
        <FONT size=1>Files</FONT></a><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM></center></BODY></HTML>}
    ]
    print {</BODY></HTML>} quit
]

; The following subroutine saves the edited file text that has been
; submitted by the edit routine above:

if submitted/6 = "save" [

    ; Save newly edited document (textarea tags replaced during the edit
    ; process are returned back to normal here and saved as they should
    ; be):

    write (to-file rejoin [what-dir submitted/8])
        (replace/all submitted/10 "<\/textarea>" "</textarea>")
    print {</BODY></HTML>} quit
]

; Run REBOL console (for file and OS operations).  This whole script was
; presented earlier in the tutorial:

if submitted/6 = "console" [
    if not exists? %rebol276 [
        print "<center>REBOL version 276 required!</center><br>"
    ]
    print {<center><a href="./sitebuilder.cgi?name=username&
        pass=password&submit=submit">Back to Sitebuilder</a></center>}
    entry-form: [
        print {
            <CENTER><FORM METHOD="post" ACTION="./sitebuilder.cgi">
            <INPUT TYPE=hidden NAME=username VALUE="username">
            <INPUT TYPE=hidden NAME=password VALUE="password">
            <INPUT TYPE=hidden NAME=subroutine VALUE="console">
            <INPUT TYPE=hidden NAME=submit_confirm
                VALUE="command-submitted">
            <TEXTAREA COLS="100" ROWS="10" NAME="contents"></TEXTAREA>
            <BR><BR>
```

```
            <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
            </FORM></CENTER></BODY></HTML>
        }
    ]
    if submitted/8 = "command-submitted" [
        write %commands.txt join "REBOL[]^/" submitted/10
        ; The "call" function requires REBOL version 2.76:
        call/output/error
            "./rebol276 -qs commands.txt"
            %conso.txt %conse.txt
        do entry-form
        print rejoin [
            {<CENTER>Output: <BR><BR>}
            {<TABLE WIDTH=80% BORDER="1" CELLPADDING="10"><TR><TD><PRE>}
            read %conso.txt
            {</PRE></TD></TR></TABLE><BR><BR>}
            {Errors: <BR><BR>}
            read %conse.txt
            {</CENTER>}
        ]
        quit
    ]
    do entry-form
]

; List existing files:

if submitted/6 = "listfiles" [

    ; Print a link to get back to the sitebuilder home page:

    print {<center><a href="./sitebuilder.cgi?name=username&
        pass=password&submit=submit">Back to Sitebuilder</a><br>}
    print {<table width=60% border=1 cellpadding="10">}
    print {<tr><td width=100%><br>}

    ; Get a list of all files in the current folder, and print
    ; a link which runs the edit subroutine on any selected file
    ; name:

    folder: sort read %.
    foreach file folder [
        print [rejoin [
            {    <a href="./sitebuilder.cgi?name=username&
                pass=password&subroutine=cleanedit&file=}
            file {">(edit)</a>    }
        ]]
        print [rejoin [
            {<a href="./} file {" target=_blank>} file {</a><br>}
        ]]
    ]
    print {<br></td></tr></table></center></BODY></HTML>}
    quit
]
```

That's it for steps 1, 4, and 5 in the outline. Step 2 is very simple. The new file creator routine was already taken care of in the code above. To create a list of existing pages that can be edited, I simply constructed links to each of the files, with a call to the "edit" subroutine in the submitted data (submitted/6 = "edit"). I only want content page files to appear in this list, so I removed all other file types from the list:

```
pages: sort read %.
```

```
dont-show-suffixs: [
    %.html %.jpg %.gif %.png %.bmp %.rip %.exe %.pdf %.cgi %.php
    %.zip %.txt %.tpl %.r %.tgz
]

; Remove file types listed above from the display:

remove-each page pages [find dont-show-suffixs (suffix? page)]

; Don't show directories either:

remove-each page pages [find to-string page "/"]

; And a few other odd files:

dont-show-files: [%rebol276 %sitemap %.ftpquota]
remove-each page pages [find dont-show-files page]

print "<hr><br>Edit Existing Pages:<br><br>"
foreach page pages [
    print rejoin [
        {<a href="./sitebuilder.cgi?name=username&pass=password&
        subroutine=edit&file=}
        to-string page {">} to-string page {</a>
        } ; <br>}
    ]
]
print {<br><br><hr>}
```

Step 3 is also very easy. I used rip.r from rebol.org to package the openwsiwyg editor - it just took a few seconds. The results are at http://re-bol.com/openwysiwyg.rip. Just "do" that file, using REBOL on any platform, and all the contents are unpacked. Here's the code that unpacks and runs it, along with a nice notice to the user:

```
if not exists? %./openwysiwyg/scripts/wysiwyg.js [
    write/binary %./openwysiwyg.rip to-binary decompress 64#{

        (compressed/embedded rip file data)

    }
    print {
        <center><table border="1" width=80% cellpadding="10"><tr><td>
        <center><strong>INITIAL SETUP: PLEASE RELOAD THIS PAGE AFTER
        FILES HAVE BEEN UNPACKED...</strong>
        </center></td></tr></table></center></BODY></HTML>
    }
    do %openwysiwyg.rip
    print {
        <center><table border="1" width=80% cellpadding="10"><tr><td>
        <center><strong>FILES HAVE BEEN UNPACKED: PLEASE RELOAD
        THIS PAGE NOW</strong>
        </center></td></tr></table></center></BODY></HTML>
    }
]
```

Here's the javascript code that integrates the WYSIWYG editor into our existing textarea editor. I just printed this on the same page as the textarea editor script:

```
{<script type="text/javascript"
```

```
src="openwysiwyg/scripts/wysiwyg.js"></script>
<script type="text/javascript">
    var full = new WYSIWYG.Settings();
    full.ImagesDir = "openwysiwyg/images/";
    full.PopupsDir = "openwysiwyg/popups/";
    full.CSSFile = "openwysiwyg/styles/wysiwyg.css";
    full.Width = "85%";
    full.Height = "250px";
    WYSIWYG.attach('all', full);
</script>}
```

I decided that I wanted to include a separate editor that did not include any WYSIWYG code. This would be useful for editing templates, code files, and any other text that doesn't need visual HTML editing. I simply included our original text editor code in a separate "subroutine" that can be called by setting the third GET value (submitted/6) to "cleanedit":

```
; non-wysiwyg edit:
if submitted/6 = "cleanedit" [
    write/append to-file rejoin [what-dir submitted/8] ""
    ; backup (before changes are made):
    cur-time: to-string replace/all to-string now/time ":" "-"
    document_text: read to-file rejoin [what-dir submitted/8]
    make-dir %edit_history
    write to-file rejoin [
        what-dir "edit_history/"
        to-string (second split-path to-file submitted/8)
        "--" now/date "_" cur-time ".txt"
    ] document_text

    prin rejoin [
        {<center><strong>Be sure to SUBMIT when done:</strong><BR><BR>
        <FORM method="post" ACTION="./sitebuilder.cgi">
        <INPUT TYPE=hidden NAME=username VALUE="} submitted/2 {">
        <INPUT TYPE=hidden NAME=password VALUE="} submitted/4 {">
        <INPUT TYPE=hidden NAME=subroutine VALUE="save">
        <INPUT TYPE=hidden NAME=path VALUE="} submitted/8 {">
        <textarea id="textarea12" name="test2" cols="100" rows="15"
            name="contents">}
        replace/all document_text "</textarea>" "<\/textarea>"
        {</textarea><br>
        <a href="./sitebuilder.cgi?name=username&pass=password&
           subroutine=listfiles-popup" target=_blank>
        <FONT size=1>Files</FONT></a><BR><BR>
        <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
        </FORM></center></BODY></HTML>}
    ]
    print {</BODY></HTML>} quit
]
```

Step 5 is the main work of this project. The first thing I did was make a link on the main page that would run a "buildsite" subroutine:

```
print {<a href="./sitebuilder.cgi?name=username&pass=password&
    subroutine=buildsite">Build Site</a>       }
```

To start building the template system, I created some basic HTML templates, compressed them, and had the script write them to files on the server:

```
; Build site:
```

```
if submitted/6 = "buildsite" [
    if not exists? %menu.tpl [
        write %menu.tpl decompress #{
        789CB556DB4EDB40107DE72B0623A456AA63C7691E00AFA5247649A484A46129
        E2A9F265B15D8C9DAE9D00ADFA41FDCBCEAEED90847015355292DD999D3D73E6
        CC1873D71EF7E8C5C4813E1D0D6172D61D0E7AA0A89A76DEEA699A4DEDD2F0B9
        A103E56E9AC7459CA56EA269CE8962ED98C268997DA7635B261DD0A163EDE063
        EEAA2AA02BF3E6711230FEBD888B8481AA5A20CD5AE96A8E1CDA81A828662AFB
        398F17A497A5054B0B95DECD18F8E5822805BB2DB4A8B84E8EC08F5C9EB382DC
        C46990DDE46AD3681B02860CD4A774A23A5FCF06DF88327143A63A18802BD01B
        9F50E7841265CA16CC4D641A1FC09E7357E4428C4FF78991A6F151C43BA51743
        07042FD5F57E9EE3BE7B18650BC6E137824B327E085E326747203CD480F95919
        F110E629269DC429DA3CD7BF0A79863B87B0F7E5C0B11D1BFE2003F206BC4993
        DCED98DDB17D015ED81371C99E2E1FDCA69D2E027193384C89CF443E2B1189A2
        809771BC8BE8E0B32499B84110A72131E4EA74E6FAE52A62711821934D5DDF57
        E0260E8A882807ED7D912A15378BEFA9F8B037C35788121120E4EC0E161D09A6
        C8664B78EF85482EDE00A93A6EB4CAD3DB3055676E2214658DB0B986B0A9BF1C
        220868880C6F9BAC15072D5F506DA53C48374110D807BF1869A1E5944EC727C7
        F8C38588B34BA2A088D96D43285BB1FAD9353335D712CAA8DC34114A7C4F7017
        2FC30F418826132CB79EA0C79397574934756B13A96CC41A49435BC5620E46C7
        95B3E7E64C0819974885D0AA3385C1A873ECAC143ACA9134863F72EE8B581173
        D1D0F8310B3116E654F6FC7A16756D9F97D35A3EFF5B5CD551437F58DAD5CA96
        3A929535ACBAB02396CE97E5ABAAB72D6DC1C6EB1AED0194AE909F24ABCC0E7B
        D9326BE3E6F0C5FA5DE562F89A5AEDB2D45085AE1ECB5321B792A1DA074C8F57
        4E0F51BFB87A7B97F279DFFA9945F02C8BF7055DF273CF4211BCB81ACB38CBF1
        29E0949AA83AFCD102CCDC22DAE0FF125F6E2B00DE21918DBE5A0F87AA457A53
        D25AE7F63513BCD97E6B976D657855DD9B7C556FFE923231E56A716E6AF3BEB3
        1EB73CF71AD90B5AE2AF4EA7BD6DA6D76DF25498F295FD449847E647D3EA65B3
        3B2E4EC15F3074FD006814E770CE3C3845AF06749204A43D07CE72C6172C686C
        1D2F2FA747125A954893FFC3EDFC039C3D1A760A0A0000
        }
    ]
    if not exists? %nomenu.tpl [
        write %nomenu.tpl decompress #{
        789CAD556D6FDA3010FECEAFB8A6AAB4490B09B47C288D235192162468197557
        F5D3E4246E92CD4B9863A0DDB41FB47FB973121874636DA7FA83E397E3B9E79E
        3B1FCE9E77D9A7B7131F06743C82C9F5E968D807C3B4AC9BC3BE6579D4AB2E8E
        9A3650C9B22255699E316159FE85E1361C7DE93A03BFE7B90E1DD291EF420387
        B3679A80B63C98A722E2F2A34A95E0609AF5B555D9369CB14F7B90283533F9D7
        79BA20FD3C533C53267D987108AB0D3114BF5756A2BE88130813260BAEC832CD
        A27C5998AD76A76DAC8006944E4CFFFDF5F00331262CE6A68F00D280FEE505F5
        2F2831A67CC19928037903DE5C321D0D69BFFB1D1A691DBDD57857F476E48356
        A6761F16059EB36E922FB884EF484EE4B20B8198F313D01666C4C3BC42ECC23C
        C3A8459AE15DC0C2CFB1CCF1A40BFB67C7BEE77BF00325283DA027AB54AFE19C
        5E7AB710C47D8D4BF6ED72E031ED9D221126D2382321D7F16C2012C3802097E8
        8BD8107221262C8AD22C26ED7277356361B54B781A27A864CBB60F0C58A6914A
        8871DC39D0A152ED597FA77AF21EC3D78C840688257F8045AF24A3F2D99ADE6B
        312A37DB9476330A041EAF705AB6EB4CB66472CB4A6390487E478CA68505C3EF
        9BBA8A0CD7198ECF6BE380155C270AB7C846E7C29FC270DC3BF73702490AE4CD
        7151C89018096778DCFC348B11C96295236B826BEAE969BA53CCA7A55BE5FFAE
        1CAF2B253C2BBD355A1B05AD8574ACF5424565B42F016AD9EB7AD38CD6A08F5B
        C48CA94477880D6F28E74BBDEDA6FDCFC4E06B463D3372B82DE64B0ABCD5F9CF
        B4FC3DC28D52F943AABA31566AD52295E1E154E2E257D3DEA8C8DD374F3DFCBA
        15ADC2E9A0EDA387E69C617FAD1A225926C813C97EE3A4E5F6F3D983D4BF829F
        D0B6ED63A0495AC00D0FE00AAD9A003D21A0342840F282CB058F9A8EA5E1DCED
        07F5FCC050885A5AABFC736AFC02A1651F4AE3060000
        }
    ]
]
```

The documents above are just simple generic HTML pages, with 4 codes inserted:

```
sitebuilder_title     ; Page title in head tag
                      ;   ** By default same as the source file name **
sitebuilder_links     ; Link menu(s) generated by this script
                      ;   (as defined in the site map that we'll create)
sitebuilder_path      ; Links back up through the hierarchy of sub-pages,
                      ;   to the home page, which we'll also create
sitebuilder_content   ; All of the data contained in the source file
                      ;   of each content page
```

Wherever those codes are found, the script will replace them with appropriate data, and then save the newly constructed files as web pages to be viewed on the site.

I'll keep the list of links that should appear on each page, along with the site's entire sub-page tree structure in a file named sitemap.r. When the script first starts up, it will create the sitemap.r file, and a blank home page content file:

```
if not exists? %sitemap.r [
    write %sitemap.r {%Home []}
    write %Home {}
]
```

Starting with the home page, every entry in the site map will simply consist of a block containing 2 items: a source file name, and a block of sub-page links. The site map must have one and only one "home" page. It can be any file name, but by default, we'll use "Home" (notice the %Home file name created above - that can be replaced by the user in the site map, but it's a recommended default). We'll also automatically create an index.html page that forwards to the home page, if no index.html exists. Here's an example of how the site map would look with only one page on the web site, labeled "Home.html":

```
%Home []
```

The file name (%Home above) contains the name of a source file to be processed (a content file that the user can upload or create with the built-in editor). The block following it (empty above) contains the names of any sub-pages that will be processed and automatically linked to it (none in the case above).

Below is an example of how the site map would look for a site made up of a home page and two sub-pages. Home.html, Page_One.html and Page_Two.html would all be created from the source files listed, and a menu bar would be automatically generated and placed on Home.html, linking to the 2 other pages. Neither Page_One.html nor Page_Two.html would contain any menu bars with links, because they don't contain any sub-pages:

```
%Home [               ; your home page (index.html forwards to it)
    [%Page_One []]    ; Page_One.html appears in the menu bar of Home.html
    [%Page_Two []]    ; Page_Two.html appears in the menu bar of Home.html
]
```

The next example site map below contains a home page with 5 sub pages, the 3rd of which contains 2 sub pages, and the 2nd of that contains 3 sub pages. In the generated .html pages, link menus are only placed on pages which have sub-pages (i.e., only Home.html, Page_Three.html and Page_Three_B.html below would contain link menus):

```
%Home [                       ; your home page
    [%Page_1 []]              ; Page_1.html appears in menu of Home.html
    [%Page_2 []]              ; Page_2.html appears in menu of Home.html
```

```
    [%Page_3 [                ; Page_3.html appears in menu of Home.html
        [%Page_3_A []]        ; Page_3_A.html is in menu of Page_3_A.html
        [%Page_3_B [          ; Page_3_B.html is in menu of Page_3_B.html
            [%Page_3_B_1 []] ; Page_3_B_1.html is in menu of Page_3_B.html
            [%Page_3_B_2 []] ; Page_3_B_2.html is in menu of Page_3_B.html
            [%Page_3_B_3 []] ; Page_3_B_3.html is in menu of Page_3_B.html
        ]]
    ]]
    [%Page_4 []]              ; Page_4.html appears in menu of Home.html
    [%Page_5 []]              ; Page_5.html appears in menu of Home.html
]
```

The key to understanding the site map idea is that any source file names followed by a link block will contain an auto-generated menu of links to those sub-pages in the .html files generated by this script. Pages without link blocks will not contain any sub-page links. They are simply wrapped in a template. Of course, users can manually link to any page they've created, if they don't want any auto-generated link menus or template design to appear on the site. They can use this script to simply upload content, or to create/edit HTML/script files. If that's the case, they don't need to create a site map at all.

To process the pages and build the site, my thought process is to create a function that reads each item in the site map and does the following:

1. If the item does not contain any sub-pages, use the nomenu.tpl template. If it does contain sub-pages, use the menu.tpl template.
2. Read the contents of the file listed, and replace the sitebuilder_content code in the template with the contents of the file.
3. Replace the sitebuilder_title code in the template with the name of the file given.
4. If the current item *is a sub-page of another page*, replace the sitebuilder_path code in the template with links to the containing pages, to create a path up through the tree structure, all the way back to the home page.
5. If the item contains sub-pages, replace the sitebuilder_links code in the template with links to each of the sub-pages listed.
6. Save the file using the file name given, plus ".html".
7. Call this function recursively for every sub-page that is contained in the current item. This will work through the entire site map.

Here's the code I came up with:

```
; First, load the sitemap and get the name of the homepage:

homepage: to-string first load %sitemap.r
current-path: rejoin [
    {<a href="./} homepage {.html">} homepage {</a>}
]

; Set up a flag variable to help determine if I'm on the home page
; during the recursion process:

begin-recurse: true

; Here's the main function.  It takes the name of a page to read
; from the site map, and a current-path variable, to keep track
; of where we currently are in the site-map tree:

recurse: func [page current-path][

    ; Set the current path (where we are in the site map):

    either begin-recurse = true [
        print-path: (to-string page/1)
```

```
] [
    print-path: rejoin [current-path { : } (to-string page/1)]
]
begin-recurse: false

; STEP 1

; Choose whether to use the menu.tpl or nomenu.tpl template
; (page/2 refers to the block of links in the current page - if
; it's empty, use the template without menus):

either (page/2 = []) [

    ; STEP 2 (for pages with no menu):

    constructed: replace (
        read %nomenu.tpl
    ) {<!-- sitebuilder_content -->} (read to-file page/1)

    ; STEP 3 (for pages with no menu):

    constructed: replace constructed {<!-- sitebuilder_title -->}
        (to-string page/1)

    ; STEP 4 (for pages with no menu):

    constructed: replace constructed {<!-- sitebuilder_path -->}
        print-path

] [

    ; STEP 2 (for pages with a menu):

    constructed: replace (read %menu.tpl)
        {<!-- sitebuilder_content -->}(read to-file page/1)

    ; STEP 5 (for pages with a menu).  This code creates an HTML link
    ; with a nice color changing rollover effect, for each sub-page
    ; listed on the current page (the page/2 block):

    link-list: copy {}
    foreach item page/2 [
        link-list: rejoin [
            link-list
            {<TR><TD style="border: solid" }
            {onmouseOver="this.bgColor='#FFFFFF'"; }
            {onmouseOut="this.bgColor='#D3D3D3'";> }
            {<CENTER><FONT face="Arial, Verdana,
                MS Sans Serif" size=1>}
            {<A HREF="./} (to-string item/1) {.html">}
                (to-string item/1) {</A>}
            {</FONT></CENTER></TD></TR>}
            newline
        ]
    ]

    ; Now add it to the menu area in the template:

    constructed: replace constructed {<!-- sitebuilder_links -->}
        link-list

    ; STEP 3 (for pages with a menu):
```

```
        constructed: replace constructed {<!-- sitebuilder_title -->}
            (to-string page/1)

        ; STEP 4 (for pages with a menu):

        constructed: replace constructed {<!-- sitebuilder_path -->}
            print-path

    ]

    ; Step 6:

    write (to-file join page/1 ".html") constructed
    print page/1 print { ... DONE<br>}

    ; Step 7:

    ; This code builds the sitebuilder path and calls the recurse
    ; function on every page in the link list:

    if not (page/2 = []) [
        if (to-string page/1) <> homepage [
            current-path: rejoin [
                current-path
                { : <a href="./} (to-string page/1) {.html">}
                (to-string page/1) {</a>}
            ]
        ]
        foreach block page/2 [recurse block current-path]
    ]
]
print {<center><table border="1" width=80% cellpadding="10"><tr><td>}

; Start the whole build process by calling the recurse function on
; the sitemap.  This will start with the home page and work down through
; the tree structure:

recurse mymap: load %sitemap.r current-path
print {</td></tr></table><br><a href="./sitebuilder.cgi?name=username&
    pass=password&submit=submit">Back to Sitebuilder</a></center>}

; Write an index.html file that forwards to the home page:

if not exists? %index.html [
    write %index.html rejoin [{
        <html>
        <head>
        <title></title>
        <META HTTP-EQUIV="REFRESH" CONTENT="0; URL=./}
        (to-string mymap/1) {.html">
        </head>
        <body bgcolor="#FFFFFF"><div id="divId">
        </div>
        </body>
        </html>
    }]
]
```

I added the following link to the sitebuilder home page to allow me to manually edit the sitemap.r file using the editor subroutine (the one without WYSIWYG):

```
print {<a href="./sitebuilder.cgi?name=username&pass=password&
```

```
    subroutine=cleanedit&file=sitemap.r">Edit Site Map</a>       }
```

At this point, I can create and edit my own sitemap.r file and actually build the site. Now I need to write the code that builds the site map *automatically*. I added the following code to the "save" subroutine created earlier, to keep users from ever having to understand the site map format, or from ever having to manually create/edit the sitemap.r file:

```
either (submitted/8 <> "sitemap.r") and (
    submitted/8 <> (to-string first load %sitemap.r)
) [
    print {<center><strong>Document Saved</strong><br><br>}

    ; This code recurses through the existing site map, and lists all
    ; the existing pages:

    recurse-sitemap: func [page] [
        append sitemap-pages page/1
        if not (page/2 = []) [
            foreach block page/2 [recurse-sitemap block]
        ]
    ]
    sitemap-pages: copy []
    recurse-sitemap load %sitemap.r
    prin {<table border="1" width=80% cellpadding="10"><tr><td>
        <center>Now ADD this page as a SUB-PAGE of another in
        your site map:<br><br>}

    ; For each page currently in the site map, print a link back to the
    ; sitebuilder script, which will run the subroutine that actually
    ; adds the new page to the selected location in the site tree:

    foreach page sitemap-pages [
        prin rejoin [
            {<a href="./sitebuilder.cgi?name=username&pass=password&
                subroutine=addsitemap&newpage=}
            submitted/8 {&existingpage=} page {">} page {</a>       }
        ]
    ]

    ; Give the user the option to not add the current page to the
    ; site map:

    print {
        <br><br>If you've ALREADY added this page to your site map,
        or if you do not want it in your site map
        <a href="./sitebuilder.cgi?name=username&pass=password&
            submit=submit"><strong>click here</strong></a>
        </center><br></td></tr></table></center>
    }
] [
    print {<html><head><META HTTP-EQUIV="REFRESH" CONTENT="0;
        URL=./sitebuilder.cgi?name=username&pass=password&
        submit=submit"></head>}
]
```

This is the subroutine called when the code above is run and submitted. It recurses through the existing site tree, finds the page that the user has selected and adds the new sub-page info, then saves the data to the sitemap.r file:

```
if submitted/6 = "addsitemap" [
```

```
    recurse-add-sitemap: func [page] [
        if page/1 = (to-file submitted/10) [
            new-block: copy []
            append new-block (to-file submitted/8)
            append/only new-block []
            insert/only page/2 new-block
        ]
        if not (page/2 = []) [foreach block page/2 [
            recurse-add-sitemap block
        ]]
    ]
    recurse-add-sitemap new-site-map: load %sitemap.r
    save %sitemap.r new-site-map
    print {
        <html><head><META HTTP-EQUIV="REFRESH" CONTENT="0;
        URL=./sitebuilder.cgi?name=username&pass=password&
        submit=submit"></head>
    }
]
```

At this point, the script is fully functional. I added one more subroutine to display help text:

```
;  Print instructions:
if submitted/6 = "instructions" [
    print {<pre>}
    print instructions: {

    ; ... HELP TEXT GOES HERE ...

    }

    print {<pre>}
    quit
]
```

To finish up, I added the following links to the main page, to display help, and to view the generated web site pages:

```
print rejoin [
    {<a href="./} (to-string first load %sitemap.r)
    {.html" target=_blank>View Home Page</a>       }
]
print {<a href="./sitebuilder.cgi?nameword&
    subroutine=instructions">Instructions</a>}
print {<br></td></tr></table></center></BODY></HTML>} quit
]
```

The following code is the complete web building CGI script. It's the longest program in this tutorial, so has been compressed to fit on this web page neatly. Unpack it and upload the sitebuilder.cgi script to your web server, along with a copy of REBOL version 2.76+, and you can use it to visually build WYSIWYG web sites, directly in your browser. Upload and edit files of any type, automatically create navigation link menus, wrap pages in design templates (2 included to get you started), run console scripts, and everything else required to create and manage complete sites. It takes just a few seconds to install and it's easy enough for absolute beginners to use immediately.

```
do http://re-bol.com/sitebuilder.r
```

10.18 Case 18 - Downloading Directories - A Server Spidering App

At one point, my notebook computer was disabled by a severe adware breakout. In an attempt to erase the troublesome files, the machine was rendered unable to boot to the operating system. I needed to copy a large number of recent data files that had not yet been backed up. Several options which didn't involve writing code were available to get this kind of job done. I could've removed the hard drive and put it in another machine, and then copied the files directly from one hard drive to another. I didn't have a hardware connector to install the laptop drive, and I didn't feel like taking the machine apart. I could've tried to reinstall the OS, and send the files across the network to be backed up on another computer. Without a system disk immediately available to restore the operating system, that wasn't convenient. I could've also potentially used a stand-alone local file transfer application (the "laplink" type), but without any serial/parallel ports, and without any OS access to provide USB support, I didn't have an application which made that option possible. Instead, I happened to have a Knoppix CD with which I was able to boot the laptop (http://www.knoppix.org/ provides the complete Linux operating system on a single free CD - it doesn't require any hard drive or any installation to run). I booted the computer to Knoppix, it found my network, and I started the Aprelium web server (http://aprelium.com/) on the laptop. Tada! Using another computer on the network, I was able to access all my files through the web server. I had access to the files at that point, but since I had literally thousands of files in hundreds of directories on the laptop, I couldn't download each one manually. Instead, I wrote a little spidering application in REBOL that did the job instantly.

To create the program in natural language, I thought about the process I would go through, and how I would click through the directory structure if I were to manually download each file:

1. Create a new destination folder on the client computer to hold the transferred files.
2. Start in the current subdirectory on the laptop (starting with the folder that held my data), and download all the files in it to the new destination directory on the client computer.
3. Create subdirectories in the destination directory on the client to mirror each folder in the current directory on the laptop.
4. Switch into each of the subdirectories on the laptop and on the client, and repeat steps 2-4 for each subdirectory.

I came up with the outline above by actually sitting down at the computer, and running though the process that I wanted to automate. I just took note of how the thought process was organized. Next, I converted the above ideas to pseudo-code descriptions of how I would accomplish the above things using code constructs:

1. Get an initial remote URL from the user. Use the built-in "request-text" function to do that. Then, create a local folder to mirror it, with a nicely formed name (only allowable Windows file name characters). Use the "replace" function to swap out unusable characters, and the "make-dir" function to create a destination folder with the cleaned up characters.
2. Since the file and directory listings are made available via a web server, I'm going to have to parse a web page for file names to download. That's easy - the web server puts "/" characters at the end of all folder listings, so anything without a "/" at the end is a file. Create a block of file names, and use a "foreach" loop to go through the list of files, using read/binary and write/binary functions to download the actual files to the destination folder.
3. I'll also need to parse the web page for folder names to create. Use another "foreach" loop to work through the block of folder names, and the "make-dir" function to create local directories with those names.
4. Create a function that changes directories on both the local and remote machines. In order to work with the correct folders, I'll need to create some variables to keep track of the directory I started in, the current local folder I'm writing to, and the current remote folder I'm reading. As I switch in and out of each directory, I'll use rejoin and replace functions to concatenate and remove the current folder names to and from the local directory and remote URL variables. Because I need to create a function that repeats both previous steps and THIS CURRENT step in every subdirectory, I'll need to enclose all three of those steps into a function, and call that function from within itself. (You've seen this recursive process of creating a function that calls itself, in the "simple search" case study. It's needed here to do the same thing in every folder, drilling down until there are no more subfolders.

The first step was straightforward. Here's the code I came up with:

```
; Get initial remote URL and create a local folder to mirror
; it, with a nicely formed name (only allowable Windows file
; name characters).

    initial-pageurl: to-url request-text/default trim {
        http://192.168.1.4:8001/4/}
    initial-local-dir: copy initial-pageurl
    replace initial-local-dir "http://" ""
    replace/all initial-local-dir "/" "_"
    replace/all initial-local-dir "\" "__"
    replace/all initial-local-dir ":" "--"
    lrf: to-file rejoin [initial-local-dir "/"]
    if not exists? lrf [make-dir lrf]
    change-dir lrf
    clf: lrf
```

Since steps 2-4 above would all be enclosed in a single function, I decided I should assign some variable words that would refer to the folders I'd be accessing: "lrf" = local-root-folder, "clf" = current-local-folder and "crfu" = current-remote-folder-url.

To begin step 2, I wrote a bit of code to do the parsing of the file and folder names on the current web page directory listing. I combined the parsing requirements from step 2 and 3 above, and decided to use the variable words "files" and "folders" to label the blocks that would contain the parsed results. Here's the code that I came up with to read and parse the contents of the current page into the usable blocks. It looks for any link (anything beginning with href=" and ending with "), and appends it to the folders block if it contains a "/" character. Anything that doesn't contain the "/" character gets appended to the file block:

```
page-data: read crfu
files: copy []
folders: copy []
parse page-data [
    any [
        thru {href="} copy temp to {"} (
            last-char: to-string last to-string temp
            either last-char = "/" [
                ; don't go upwards through the folder structure:
                if not temp = "../" [
                    append folders temp
                ]
            ][
                append files temp
            ]
        )
    ] to end
]
```

To complete step 2, here's the foreach loop that I came up with to download all the files contained in the file block. It contains a replace/rejoin trick to make sure the filename gets concatenated to the current URL correctly (with no extra "/"s):

```
foreach file files [
    print rejoin ["Getting: " file]
    new-page: rejoin [crfu "//" file]
    replace new-page "///" "/"
    write/binary to-file file read/binary to-url new-page
```

```
]
```

I ran into some problems with certain links on the web page that weren't actually file or folder listings, or which didn't download properly. I used some conditional "if"s and "error? try" combinations to eliminate those problems. I wrote the errors to a text file, so that I could check them afterwards and download manually if necessary. Here's the revised version of the code above, with the error handling routines:

```
foreach file files [
    if not file = "http://www.aprelium.com" [
    ; The free aprelium server puts that link on all pages
    ; it serves.  I didn't want to spider all the contents of
    ; their web page.
        print rejoin ["Getting: " file]
        new-page: rejoin [crfu "//" file]
        replace new-page "///" "/"
        if not exists? to-file file [
            either error? try [read/binary to-url new-page] [
                write/append %/c/errors.txt rejoin [
                    "There was an error reading:  "    new-page
                    newline]
            ] [
            if error? try [
            write/binary to-file file read/binary to-url new-page
            ][
                write/append %/c/errors.txt rejoin [
                "error writing: " crfu newline]]
            ]
        ]
    ]
]
```

I wanted to complete step 3, but realized that that's where the recursion pattern needed to occur - for each folder I copied, I wanted to look inside that folder and create any folders it contained, and then inside those folders, etc. So next, I defined a recursion pattern to change into the current local and remote folders, and to run the function in which all of steps 2-4 were contained. I decided to label the entire enclosing function "copy-current-dir" - it would be passed the parameters "lrf", "clf", and "crfu". That function contains the recurse function, which calls the encompassing copy-current-dir function, which itself contains the recurse function, etc. The effect of this recursion is that every subfolder of every folder is entered. Here's the recurse function:

```
recurse: func [folder-name] [
    change-dir to-file folder-name
    crfu: rejoin [crfu folder-name]
    clf: rejoin [clf folder-name]
    ; NOW HERE'S THE RECURSION - call the function in which
    ; this function is contained:
    copy-current-dir crfu clf lrf
    ;  When done, go back up a folder on both the local and
    ;  remote machines.  The replace actions remove the current
    ;  folder text from the end of the current folder strings.
    change-dir %..
    replace clf folder-name ""
    replace crfu folder-name ""
]
```

Finally, I completed steps 3 and 4 by creating local folders to mirror each directory in the current remote folder, and then called the recurse function to spider down through them. I used a foreach loop to work through each directory in the current subdirectory list. Because this loop contains the

recurse function, which in turn runs the copy-current-dir, which in turn contains this loop, every subdirectory of every subdirectory is worked through, until the job is complete:

```
foreach folder-name folders [
    make-dir to-file folder-name
    recurse folder-name
]
```

I wrapped the parsing, looping/reading, and recursing sections into the copy-current-dir function so that they could be called recursively. Then I added some error handling routines as I played with the working code. I included a block of URLs to be avoided, and some code in the final foreach loop to check that those URLs weren't already downloaded (in case I had previously run the program on the same directory). Here's the final script:

```
REBOL [title: "Directory Downloader"]

avoid-urls: [
    "/4/DownLoad/en_wikibooks_org/skins-1_5/common/&"
    "DownLoad/groups_yahoo_com/group/Join%20This%20Group!/"
    "DownLoad/pythonide_stani_be/ads/"
    "Nick%20Antonaccio/Desktop/programming/api/ewe/"
]

copy-current-dir: func [
{
    Download the files from the current remote directory
    to the current local directory and create local subfolders
    for each remote subfolder.  Then recursively do the same
    thing inside each sub-folder.
}
    crfu  ; current-remote-folder-url
    clf   ; current-local-folder
    lrf   ; local-root-folder
] [
    ; Check that the URL about to be parsed is not in the avoid
    ; list above.  This provides a way to skip specified folders
    ; if needed:

    foreach avoid-url avoid-urls [
        if find crfu avoid-url [return "avoid"]
    ]

    ; First, parse the remote folder for file and folder names.
    ; Given the URL of a remote page, create 2 list variables.
    ; files:  remote files to download (in current directory)
    ; folders:  remote sub-directories to recurse through.
    ; There's an error check in case the page can't be read:

    if error? try [page-data: read crfu] [
        write/append %/c/errors.txt rejoin [
            "error reading (target read error): "
            crfu newline]
        return "index.html"
    ]

    ; if the web server finds an index.html file in the folder
    ; it will serve its contents, rather than displaying the
    ; directory structure.  Then it'll try to spider the HTML
    ; page.  The following will keep that error from occuring.
    ; NOTE:  this error was more effectively stopped by
```

```
; editing the index page names in the Abyss web server:
if not find page-data {Powered by <b><i>Abyss Web Server} [
    ; </i></b>
    write/append %/c/errors.txt rejoin [
        "error reading (.html read error): "
        crfu newline]
    return "index.html"
]
files: copy []
folders: copy []
parse page-data [
    any [
        thru {href="} copy temp to {"} (
            last-char: to-string last to-string temp
            either last-char = "/" [
            ; don't go upwards through the folder structure:
                if not temp = "../" [
                    append folders temp
                ]
            ][
                append files temp
            ]
        )
    ] to end
]

; Next, download the files in the current remote folder
; to the current local folder:

foreach file files [
    if not file = "http://www.aprelium.com" [
        print rejoin ["Getting: " file]
        new-page: rejoin [crfu "//" file]
        replace new-page "///" "/"
        if not exists? to-file file [
            either error? try [read/binary to-url new-page][
                write/append %/c/errors.txt rejoin [
                    "There was an error reading:  "    new-page
                    newline]
            ] [
            if error? try [
    write/binary to-file file read/binary to-url new-page
            ][
                write/append %/c/errors.txt rejoin [
                "error writing: "
                crfu newline]]
            ]
        ]
    ]
]

; Check to see if there are no more subfolders.  If so,
; exit the copy-current-dir function

if folders = [] [return none]

; Define the recursion pattern.  This changes into the
; current local folder, and runs the copy-current-dir
; function (the current function we are in), which itself
; contains the recurse function, which itself will call
; the copy-current-dir, etc.  The effect of this recursion
; is that every subfolder of every folder is entered.
; This is what enables the spidering:
```

```
    recurse: func [folder-name] [
        change-dir to-file folder-name
        crfu: rejoin [crfu
            folder-name]
        clf: rejoin [clf
            folder-name]
        copy-current-dir crfu clf lrf
        ;  When done, go back up a folder on both the local
        ;  and remote machines.  The replace actions remove
        ;  the current folder text from the end of the current
        ;  folder strings.
        change-dir %..
        replace clf folder-name ""
        replace crfu folder-name ""
    ]

    ; Third, create local folders to mirror each directory in
    ; the current remote folder, and then spider down through
    ; them using the recurse function to download all the files
    ; and subdirectories included in each folder:

    foreach folder-name folders [
    ;    foreach avoid-url avoid-urls [
    ;        if not find folder-name avoid-url [
                make-dir to-file folder-name
                recurse folder-name
    ;        ]
    ;    ]
    ]
]

; Now, get initial remote URL and create a local folder to
; mirror it, with a nicely formed name (only allowable Windows
; file name characters).

    initial-pageurl: to-url request-text/default trim {
        http://192.168.1.4:8001/4/}
    initial-local-dir: copy initial-pageurl
    replace initial-local-dir "http://" ""
    replace/all initial-local-dir "/" "_"
    replace/all initial-local-dir "\" "__"
    replace/all initial-local-dir ":" "--"
    lrf: to-file rejoin [initial-local-dir "/"]
    if not exists? lrf [make-dir lrf]
    change-dir lrf
    clf: lrf

; Start the process by running the copy-current-dir function:

copy-current-dir initial-pageurl clf lrf

print "DONE" halt
```

10.19 Case 19 - Vegetable Gardening

My mother is a retired Microsoft Access developer who loves to garden in her spare time. She's collected a wide scope of knowledge about how certain plants survive better when planted next to each other, and she wanted to create a program to help organize that info. She wanted to create a standalone version that she could use on her home computer and give to friends. She also wanted to publish it to the web as a dynamic database. Additionally, she anticipated creating a version that could be carried into the garden on a pocket pc. I suggested using REBOL, because it could provide a solution for all her needs. She'd been working for several days with her development tools, and I told her I could get the whole thing done that same evening using REBOL. Here's the outline I created:

1. Create a database structure to hold the vegetable compatibility info and other related information.
2. Write a command line version of the script that allows users to display all the info for any selected vegetable (this could be run on any operating system that supports the command line version of REBOL, including pocket pc).
3. Create a CGI version of the above script that works on the web site.
4. Create a pretty GUI version to be used on the home PC.
5. Write a separate GUI to manage the administrative adding of data to the database.
6. Provide a way to update the data files on the web site.

To get things started, I used the listview data grid example from this tutorial to provide a front end for the vegetable data files. This provided a data structure that was suitable for the project, and it formed an instant solution to creating a GUI front end. Steps 1 and 5 were instantly completed (that database example is so useful - many thanks to Henrik Mikael Kristensen for creating the listview module!).

I created a few initial lines of data to work with. Here's the working database.db file that I created:

```
["basil" "" "tomato" "basil protects tomatoes." "" ""]
["beans" "onion" "cabbage carrot radish" "" "" ""]
["cabbage" "celery" "tomato" "" "" ""]
["carrot" "" "tomato"
    "Carrots strengthen the roots of tomatoes."
    "Carrots love tomatoes." ""]
["radish" "cabbage" "beans carrot tomato" "" "" ""]
["tomato" "cabbage" "basil carrot" "" "" ""]
```

Each block holds 6 pieces of information about each possible vegetable:

1. the name of the veggie
2. a list of other veggies that are compatible with the given veggie (those that do well when planted next to the given veggie).
3. a list of other veggies that are incompatible
4. 3 fields for general notes about the given veggie

I decided to add an "upload" button to the listview GUI to satisfy step #6 in my program outline. It made sense to add this functionality here, because the user workflow would generally involve adding/changing data in the database (using the listview), and then updating the online database to match. Here's the upload code I came up with. It includes some error checking, so that the application doesn't crash if there's a problem with the Internet connection. I added a button to the listview GUI and put the above code in its action block. Here's the complete code I added to the listview:

```
btn "upload to web" [
    uurl: ftp://user:pass@website.com/public_html/path/
    if error? try [
        ; first, backup former data file:
```

```
        write rejoin [uurl "database_backup.db"] read rejoin [
            uurl "database.db"]
        write rejoin [uurl "database.db"] read %database.db
        alert "Update complete."
    ] [alert "Error - check your Internet connection."]
]
```

Next, I realized that adding and removing new vegetables to and from the database would require some special consideration. It ended up being the biggest part of this coding project. I could use the built-in abilities of the listview module to simply add a new vegetable to the database, but there was a problem with that. Every time a new vegetable is added to the database, it creates a list of compatibilities. Aside from simply adding a new block to the database with fields listing the compatibilities and incompatibilities, that new veggie needs to be added to the compatibility list of every other vegetable with which it's compatible. It also needs to be added to the incompatibility list of every vegetable with which it's not compatible. Editing those blocks manually would take a lot of work and introduce a greater likelihood for user errors, especially as the database grows larger. Instead, I decided to create a little script to do it automatically. Here's the pseudo code thought process for that script:

1. Create a list of existing vegetables. This can be done by reading the existing database, looping through each block, and picking out the first item in each block (the vegetable name).
2. Create a small new GUI to enter the new veggie info. It should include an input field for the new veggie name, 2 text-lists showing the possible compatible and incompatible veggies (read from the existing list of veggies in the database), and 3 note fields.
3. Use a foreach loop to run through the lists of compatible and incompatible veggies. Have the loop automatically add the new vegetable to the other veggies' respective compatibility lists.

I created the GUI code and put the foreach loop inside the action block of a button used to add the new veggie. Here's the code, which I saved as "add_veggie.r":

```
REBOL [title: "Add Veggie"]

; read the current database:
veggies: copy load %database.db
; get the list of veggies (the 1st item in each block):
veggie-list: copy []
foreach veggie veggies [append veggie-list veggie/1]

; create a GUI with the appropriate fields and text-lists:
view/new center-face add-gui: layout [
    across
    text "new vegetable:" 88x24 right new-veg: field
    return
    text "compatible:" 88x24 right
    new-compat: text-list data veggie-list
    return
    text "incompatible:" 88x24 right
    new-incompat: text-list data veggie-list
    return
    text "note 1:" 88x24 right new-note1: field
    return
    text "note 2:" 88x24 right new-note2: field
    return
    text "note 3:" 88x24 right new-note3: field
    return

    ; now add a button to run the foreach loops:

    tabs 273 tab btn "Done" [
        ; First, append the new veggie data block to
        ; the existing database block.  Create the new
```

```
        ; block from the text typed into each field,
        ; and from the items picked in each of the
        ; lists above ("reduce" evaluates the listed
        ; items, rather than including the actual text.
        ; i.e., you want to add the text typed into the
        ; new-veg field, not the actual text
        ; "new-veg/text").  "append/only" appends the
        ; new block to the database as a block, rather
        ; than as a collection of single items:
        append/only veggies new-block: reduce [
            new-veg/text
            ; "reform" creates a quoted string from the
            ; block of picked items in the text-lists:
            reform new-compat/picked
            reform new-incompat/picked
            new-note1/text
            new-note2/text new-note3/text
        ]
        ; Now loop through the compatibility list of the
        ; new veggie, and add the new veggie to the
        ; compatibility lists of all those other
        ; compatible veggies.  I put a space in if there
        ; were already other veggies in the list:
        foreach onecompat new-compat/picked [
            foreach veggie veggies [
                if find veggie/1 onecompat [
                    either veggie/2 = "" [spacer: ""] [
                        spacer: " "]
                    append veggie/2 rejoin [spacer
                    new-veg/text]
                ]
            ]
        ]
        ; Now do the same thing for the incompatibility
        ; list:
        foreach oneincompat new-incompat/picked [
            foreach veggie veggies [
                if find veggie/1 oneincompat [
                    either veggie/3 = "" [spacer: ""] [
                        spacer: " "]
                    append veggie/3 rejoin [spacer
                    new-veg/text]
                ]
            ]
        ]
        save %database.db veggies
        ; start the veggie data editor again when done:
        launch %veggie_data_editor.r
        unview add-gui
    ]
]
focus new-veg
do-events
```

Because the add_veggie.r script will always be run from the veggie_data_editor.r program, I added the following code to the action block for the "add veggie" button in the data editor. It launches the above add_veggie program, and closes the listview:

```
btn "add veggie" [launch %add_veggie.r quit]
```

When the user closes the add_veggie program, the "launch %veggie_data_editor.r" code at the end

of the program relaunches the data editor. This handles flipping back and forth between the two screens. When the data editor is relaunched, all the new data is automatically updated and displayed, so I don't need to manually update any displayed info. After playing with the system, I realized before closing the data editor I'd better save the changes made to the database. So I adjusted the above code as follows:

```
btn "add veggie" [
    launch %add_veggie.r
    backup-file: to-file rejoin ["backup_" now/date]
    write backup-file read %database.db
    save %database.db theview/data
    quit
]
```

Next, I used the above code to create a similar "remove_veggie.r" program. Instead of building a GUI for it, I just added some code to the "remove veggie" button in the veggie data editor to save the name of the currently selected vegetable to a file (veggie2remove.r). I also copied the backup routine from the code above to make sure any changes in the listview are saved before going on:

```
btn "remove veggie" [
    if (to-string request-list "Are you sure?"
            [yes no]) = "yes" [
        ; get the veggie name from the currently selected
        ; row in the listview:
        first-veg: copy first theview/get-row
        theview/remove-row
        write %veggie2remove.r first-veg
        launch %remove_veggie.r
        backup-file: to-file rejoin ["backup_" now/date]
        write backup-file read %database.db
        save %database.db theview/data
        quit
    ]
]
```

The remove_veggie.r script just reads the vegetable name from the veggie2remove.r file created above, and runs through some foreach loops to delete that vegetable from the compatibility lists of the other veggies:

```
REBOL [title: "Remove Veggie"]

veggies: copy load %database.db
remove-veggie: read %veggie2remove.r

; remove the selected veggie from compatible lists (the second
; field in each block).  This is done by replacing any
; occurrence of the remove-veggie with an empty string ("").
; That effectively erases every occurrence of the veggie:

foreach veggie veggies [
    replace veggie/2 remove-veggie ""
]

; do the same thing to the incompatible lists of all other
; veggies (field 3 in each block):

foreach veggie veggies [
    replace veggie/3 remove-veggie ""
```

```
]

save %database.db veggies
; start the veggie data editor again when done:
launch %veggie_data_editor.r
```

Now the listview data editor and all its helper scripts are complete. Because the listview is generally run from the GUI version of the main program ("veggie_gui.r" - not yet written), I added the following code to the existing listview close routine:

```
launch "veggie_gui.r"
```

When I design the main veggie_gui program, I'll add a button to launch the listview. When I close the listview, the above code will relaunch the GUI program to handle flipping back and forth between those two screens. Here's the final listview data grid code with all the described changes and additions:

```
REBOL [title: "Veggie Data Editor"]

evt-close: func [face event] [
    either event/type = 'close [
        inform layout [
            across
            btn "Save Changes" [
                ; when the save btn is clicked, a backup data
                ; file is automatically created:
                backup-file: to-file rejoin ["backup_" now/date]
                write backup-file read %database.db
                save %database.db theview/data
                launch "veggie_gui.r"
                quit
            ]
            btn "Lose Changes" [
                launch "veggie_gui.r"
                quit
            ]
            btn "CANCEL" [hide-popup]
        ] none ] [
        event
    ]
]
insert-event-func :evt-close

if not exists? %list-view.r [write %list-view.r read
    http://www.hmkdesign.dk/rebol/list-view/list-view.r
]
do %list-view.r

if not exists? %database.db [write %database.db {[][]}]
database: load %database.db

view center-face gui: layout [
    h3 {To enter data, double-click any row, and type directly
        into the listview.  Click column headers to sort:}
    theview: list-view 775x200 with [
        data-columns: [Vegetable Yes No Note1 Note2
            Note3]
        data: copy database
        tri-state-sort: false
```

```
        editable?: true
    ]
    across
    btn "add veggie" [
        launch %add_veggie.r
        backup-file: to-file rejoin ["backup_" now/date]
        write backup-file read %database.db
        save %database.db theview/data
        quit
    ]
    btn "remove veggie" [
        if (to-string request-list "Are you sure?"
                [yes no]) = "yes" [
            first-veg: copy first theview/get-row
            theview/remove-row
            write %veggie2remove.r first-veg
            launch %remove_veggie.r
            backup-file: to-file rejoin ["backup_" now/date]
            write backup-file read %database.db
            save %database.db theview/data
            quit
        ]
    ]
    btn "filter veggies" [
        filter-text: request-text/title trim {
            Filter Text (leave blank to refresh all data):}
        theview/filter-string: filter-text
        theview/update
    ]
    btn "upload to web" [
        uurl: ftp://user:pass@website.com/public_html/path/
        if error? try [
            ; first, backup former data file:
            write rejoin [
                uurl "database_backup.db"] read rejoin [
                uurl "database.db"]
            write rejoin [uurl "database.db"] read %database.db
            alert "Update complete."
        ] [alert "Error - check your Internet connection."]
    ]
]
```

Next, I created a command line version of the program. The "Looping Through Data" example provided earlier in this tutorial served as a perfect model. I just changed some of the variable labels and loaded the data from the existing database.db file. Here's the code:

```
REBOL [title: "Veggies"]

veggies: load %database.db

a-line: copy [] loop 65 [append a-line "-"]
a-line: trim to-string a-line

print-all: does [
    foreach veggie veggies [
        print a-line
        print rejoin ["Veggie:   " veggie/1]
        print a-line
        print rejoin ["Matches:  " veggie/3]
        print rejoin ["No-nos:   " veggie/2]
        print rejoin ["Note 1:   " veggie/4]
```

```
        print rejoin ["Note 2:   " veggie/5]
        print rejoin ["Note 3:   " veggie/6]
        print newline
    ]
]
forever [
    prin "^(1B)[J"
    print "Here are the current foods in the database:^/"
    print a-line
    foreach veggie veggies [prin rejoin [veggie/1 "  "]]
    print "" print a-line
    print "Type a vegetable name below.^/"
    print "Type 'all' for a complete database listing."
    print "Press [Enter] to quit.^/"
    answer: ask {What food would you like info about?  }
    print newline
    switch/default answer [
        "all"     [print-all]
        ""         [ask "^/Goodbye!  Press [Enter] to end." quit]
        ][
        found: false
        foreach veggie veggies [
            if find veggie/1 answer [
                print a-line
                print rejoin ["Veggie:   " veggie/1]
                print a-line
                print rejoin ["Matches:  " veggie/3]
                print rejoin ["No-nos:   " veggie/2]
                print rejoin ["Note 1:   " veggie/4]
                print rejoin ["Note 2:   " veggie/5]
                print rejoin ["Note 3:   " veggie/6]
                print newline
                found: true
            ]
        ]
        if found <> true [
            print "That vegetable is not in the database!^/"
        ]
    ]
    ask "Press [ENTER] to continue"
]
halt
```

That was easy! Just compare it to the original example - it's virtually identical. Again, that generalized example was presented in this tutorial to provide a model for use in many varied situations. Using it, I didn't even need to write any pseudo code.

Now I extended the above command line example to create a CGI application. To get started, I used the final CGI example provided earlier in this tutorial as a model. To it, I added the code that I'd created for the command line example above. The only real changes I needed to make were some additional HTML formatting tags, required make the page display properly in a browser (mostly newline "< B R >"s). Again, just an amalgam of several existing examples. No pseudo code required - I just had to think about how to arrange the existing command line code to fit into the general CGI outline. Here's the code:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "Veggies"]
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Veggies"</TITLE></HEAD><BODY>]

veggies: load %database.db
```

```
a-line: copy [] loop 65 [append a-line "-"]
a-line: trim to-string a-line

print-all: does [
    foreach veggie veggies [
        print a-line
        print [<BR>]
        print rejoin ["Veggie:   " veggie/1]
        print [<BR>]
        print a-line
        print [<BR>]
        print rejoin ["Matches:  " veggie/3]
        print [<BR>]
        print rejoin ["No-nos:   " veggie/2]
        print [<BR>]
        print rejoin ["Note 1:   " veggie/4]
        print [<BR>]
        print rejoin ["Note 2:   " veggie/5]
        print [<BR>]
        print rejoin ["Note 3:   " veggie/6]
        print [<BR>]
    ]
]

print "Here are the current foods in the database:^/"
print [<BR>]
print a-line
print [<BR><strong>]
foreach veggie veggies [prin rejoin [veggie/1 "  "]]
print ""
print [</strong><BR>]
print a-line
print [<BR>]

submitted: decode-cgi system/options/cgi/query-string
if submitted/2 <> none [
    switch/default submitted/2 [
        "all"     [print-all]
        ][
        found: false
        foreach veggie veggies [
            if find veggie/1 submitted/2 [
                print a-line
                print [<BR>]
                print rejoin ["Veggie:   " veggie/1]
                print [<BR>]
                print a-line
                print [<BR>]
                print rejoin ["Matches:  " veggie/3]
                print [<BR>]
                print rejoin ["No-nos:   " veggie/2]
                print [<BR>]
                print rejoin ["Note 1:   " veggie/4]
                print [<BR>]
                print rejoin ["Note 2:   " veggie/5]
                print [<BR>]
                print rejoin ["Note 3:   " veggie/6]
                found: true
            ]
        ]
        if found <> true [
            print [<BR>]
```

```
            print "That vegetable is not in the database!"]
            print [<BR>]
    ]
]

print [<FORM ACTION="http://website.com/rebol/veggie.cgi">]
print [<BR><HR><BR>"Enter a veggie you'd like info about:"<BR>]
print ["Veggie: "<INPUT TYPE="TEXT" NAME="username" SIZE="25">]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</BODY></HTML>]
```

I didn't like the way the CGI required the user to type in the name of a listed vegetable. Instead, I got rid of the list printout, and added the list to a selectable dropdown box. Here's the final cgi example with the HTML dropdown box:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "Veggies"]
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Veggies"</TITLE></HEAD><BODY>]

veggies: load %database.db

a-line: copy [] loop 65 [append a-line "-"]
a-line: trim to-string a-line

print-all: does [
    foreach veggie veggies [
        print a-line
        print [<BR>]
        print rejoin ["Veggie:   " veggie/1]
        print [<BR>]
        print a-line
        print [<BR>]
        print rejoin ["Matches:  " veggie/3]
        print [<BR>]
        print rejoin ["No-nos:   " veggie/2]
        print [<BR>]
        print rejoin ["Note 1:   " veggie/4]
        print [<BR>]
        print rejoin ["Note 2:   " veggie/5]
        print [<BR>]
        print rejoin ["Note 3:   " veggie/6]
        print [<BR>]
    ]
]

submitted: decode-cgi system/options/cgi/query-string
if submitted/2 <> none [
    switch/default submitted/2 [
        "all"     [print-all]
        ][
        found: false
        foreach veggie veggies [
            if find veggie/1 submitted/2 [
                print a-line
                print [<BR>]
                print rejoin ["Veggie:   " veggie/1]
                print [<BR>]
                print a-line
                print [<BR>]
```

```
                print rejoin ["Matches:  " veggie/3]
                print [<BR>]
                print rejoin ["No-nos:   " veggie/2]
                print [<BR>]
                print rejoin ["Note 1:   " veggie/4]
                print [<BR>]
                print rejoin ["Note 2:   " veggie/5]
                print [<BR>]
                print rejoin ["Note 3:   " veggie/6]
                found: true
            ]
        ]
        if found <> true [
            print [<BR>]
            print "That vegetable is not in the database!"]
            print [<BR>]
    ]
]

print [<FORM ACTION="http://website.com/rebol/veggie.cgi">]
print [<BR>"Please select a veggie you'd like info about:"<BR>]
print ["Veggie: "<select NAME="username"><option>"all"]
foreach veggie veggies [prin rejoin ["<option>" veggie/1]]
print [</option>]
print [<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">]
print [</FORM>]
print [</BODY></HTML>]
```

The final part of the outline that I needed to address was the GUI display version of the program. I needed to create this from scratch, so I came up with an outline and some pseudo code to organize my thoughts:

1. Display the complete list of vegetables in the database (build the list using a foreach loop similar to the ones used in the command line program, and display that block in a text list widget).
2. Display the info for any vegetable selected from the text list widget (when an item is selected, collect all the info for the selected vegetable and display it, nicely formatted, in a separate text area widget).
3. Add a button to run the listview editor created earlier.

First I borrowed some code from the add_veggies.r example to create a list of all the veggies in the database. It uses a foreach loop to cycle through each block in the database, and creates a list of the first item in each block (the name of each vegetable). Then it sorts the list alphabetically. This should be run before the GUI is displayed:

```
load-data: does [
    veggies: copy load %database.db
    veggie-list: copy []
    foreach veggie veggies [append veggie-list veggie/1]
    veggie-list: sort veggie-list
]
```

I decided to use a text-list widget to display the block of vegetable names. To display the info for each vegetable, I used a simple text area display. Here's the REBOL layout code to do that:

```
list-veggies: text-list 200x400 data veggie-list
display: area "" 300x400
```

To that text-list widget's action block I added some code to display the info about the selected

vegetable (it gets evaluated whenever the user selects an item from the list):

```
; First, build a block of text with all the info about the
; selected vegetable, nicely formatted with newlines and
; capitalized section headings:

current-info: []
foreach veggie veggies [
    if find veggie/1 value [
        current-info: rejoin [
            "COMPATIBLE:     " veggie/3 newline newline
            "INCOMPATIBLE:   " veggie/2 newline newline
            "NOTE 1:    " veggie/4 newline newline
            "NOTE 2:    " veggie/5 newline newline
            "NOTE 3:    " veggie/6
        ]
    ]
]

; Now display and update that text in the text area widget:

display/text: current-info
show display show list-veggies
```

Finally, add a button to run the listview data editor:

```
btn "Edit Tables" [do %veggie_data_editor.r]
```

That's basically it. Here's the final version:

```
REBOL [title: "Veggie Matches"]

load-data: does [
    veggies: copy load %database.db
    veggie-list: copy []
    foreach veggie veggies [append veggie-list veggie/1]
    veggie-list: sort veggie-list
]

load-data

view display-gui: layout [
    h2 "Click a veggie name to display matches and other info:"
    across
    list-veggies: text-list 200x400 data veggie-list [
        current-info: []
        foreach veggie veggies [
            if find veggie/1 value [
                current-info: rejoin [
                    "COMPATIBLE:     " veggie/3 newline newline
                    "INCOMPATIBLE:   " veggie/2 newline newline
                    "NOTE 1:    " veggie/4 newline newline
                    "NOTE 2:    " veggie/5 newline newline
                    "NOTE 3:    " veggie/6
                ]
            ]
        ]
        display/text: current-info
        show display show list-veggies
```

```
    ]
    display: area "" 300x400 wrap
    return
    btn "Edit Tables" [
        do %veggie_data_editor.r
        ; launch "veggie_data_editor.r"
        ; load-data
        ; show list-veggies
        ; show display
    ]
]
```

There are 5 complete local script files that make up the completed veggie program: veggie_data_editor.r, add_veggie.r, remove_veggie.r, veggie_command_line.r, veggie_gui.r. In general, the main desktop applications are started by running the veggie_gui.r script. The veggie_data_editor.r can also be run by itself (remember that it runs the veggie_gui.r program when it closes). In order for the veggie_data_editor to work, the listview.r file needs to be included in the same directory. The created database.db should also be kept in the same directory. I packed all those files into an executable using XpackerX, and sent it to my Mom. The 6th script file, veggie.cgi, got uploaded to the web site. The database.db file was also uploaded manually, but my Mom prefers using the upload button in the veggie_data_editor to update the database on the web site. The veggie2remove.r and database backup files are created automatically when the program is used - they're found in the same folder as the script files.

10.20 Case 20 - Coding a Freecell Game Clone (GUI)

As far as I know, there's no existing Freecell game implemented in REBOL, and it's my other favorite computer game. This project will provide some more food for thought about useful GUI techniques and approaches. Here's my initial outline:

1. Get the card images compressed and embedded into REBOL code.
2. Write the code to display and move cards around the screen. It will be similar to that found in the Guitar Chord Diagram Maker example presented earlier. I'll need to click and drag images around the screen. I'll also want to make the images "snap" into position onto other cards, rather than floating freely.
3. Create a nice looking GUI layout backdrop for the playing field.
4. Layout the cards in random order, in 8 piles, on the playing field.
5. Allow the selection and movement of cards, based on the rules of Freecell (i.e., cards need to be placed in descending order, red-black-red-black, goal piles must start with aces, and ascend through a single suit, etc.). These rules can be handled by a series of conditional evaluations that are run every time a card is moved. This step will require the most coding thought and will likely need a sub-outline.

To get started with the first step, I remembered seeing a REBOL card game at http://www.rebolfrance.org/articles/bridge/bridge.html . The zip package at that location contains all the .bmp card images in a single directory. I downloaded the package and wrote a little variation of the binary resouce embedder provided earlier in this tutorial. It loops through all the cards in the directory, reads and compresses the files, and then appends each unit of data to a single block labeled "cards", which is created to hold all the images:

```
REBOL [Title: "Card Image Embedder"]

system/options/binary-base: 64
cards: copy []
foreach file load %./  [
    uncompressed: read/binary file
    compressed: compress to-string uncompressed
    ; there are some other files in the directory that I don't
    ; want to embed.  Limiting the file size to 10k weeds them out:
    if ((length? uncompressed) < 10000) [
        append cards compressed
    ]
]
editor cards
```

Because the cards are read in alphabetical order from the directory, I need to change the order of the card data so that they ascend in the following order: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K. I also added some comments to clarify where each suit begins and ends. This provides a nice chunk of data that I can use to build other card games of any type:

```
do http://re-bol.com/cards.r
```

Next, I wrote a little GUI app to test that all the cards display appropriately. It builds a GUI block by reading, decompressing, and appending the data in the card block above, using the built-in "image" word to display each decompressed card. That GUI block is then viewed with the typical "view layout" code. All of the code in the rest of this section will assume that the card data above has been defined in the interpreter:

```
; I want the cards to be layed out next to each other in the GUI, so
; the layout block starts with the built-in "across" word:
```

```
gui: [across ]

; The "count" variable is used to separate each suit onto different
; lines in the GUI:

count: 0

foreach card cards [
    if count = 13 [
        ; after the 13th card, start a new line in the GUI
        ; and reset the count:
        append gui [return]
        count: 0
    ]
    ; The following code adds the image data to the block that'll be
    ; displayed:
    append gui compose [image load to-binary decompress (card)]
    count: count + 1
]

view layout gui
```

That provides a fundamental way to compress and reuse card images to create all types of games. Adding the "feel movestyle" code presented earlier in this tutorial allows us to click and drag the cards around the GUI. Here I'll make some changes to the code because I want the cards to move using "snap-to" positioning, as if they're placed on a grid and clicking from one grid position to the next (as opposed to floating freely across the screen with each click-drag):

```
; The following function enables the pieces to slide around the
; screen.  The coordinates are rounded to multiples of 79 and 104
; pixels to enable snap-to positioning (the horizontal width and
; vertical height of each card and the surrounding space).  The
; additional 20 pixels accounts for the default border around the
; overall GUI:

movestyle: [
    engage: func [face action event] [
        if action = 'down [
            face/data: event/offset
            remove find face/parent-face/pane face
            append face/parent-face/pane face
        ]
        if find [over away] action [
            unrounded-pos: (face/offset + event/offset - face/data)
            snap-to-x: (round/to first unrounded-pos 79) + 20
            snap-to-y: (round/to second unrounded-pos 104) + 20
            face/offset: to-pair rejoin [snap-to-x "x" snap-to-y]
        ]
        show face
    ]
]

; Here's a revised version of the previous GUI block.  The only
; difference is that it uses the snap-to positioning definition
; above ("movestyle"):

gui: [across ]
count: 0
foreach card cards [
    if count = 13 [
        append gui [return]
```

```
        count: 0
    ]
    append gui compose [
        image load to-binary decompress (card) feel movestyle
    ]
    count: count + 1
]
view layout gui
```

Now you can pick up any card, move it around the screen, and it automatically lines up and snaps over any other card on the screen - very useful! Steps 1 and 2 in the program outline are done. Next, I'll remove the card backside image from the card data, and tile all the images on the screen. That just means changing the initial positions of the cards, and also the number of pixels rounded in the snap-to code:

```
movestyle: [
    engage: func [face action event] [
        if action = 'down [
            face/data: event/offset
            remove find face/parent-face/pane face
            append face/parent-face/pane face
        ]
        if find [over away] action [
            unrounded-pos: (face/offset + event/offset - face/data)
            snap-to-x: (round/to first unrounded-pos 79) + 20
            snap-to-y: (round/to second unrounded-pos 30) + 20
            face/offset: to-pair rejoin [snap-to-x "x" snap-to-y]
        ]
        show face
    ]
]

gui: [across ]
count: 0
ypos: 20
foreach card cards [
    if count = 8 [
        ypos: ypos + 30
        coord: to-pair rejoin [20 "x" to-string compose (ypos)]
        append gui compose [at (coord)]
        count: 0
    ]
    append gui compose [
        image load to-binary decompress (card) feel movestyle
    ]
    count: count + 1
]
view layout gui
```

That code is really starting to look and act like a card game :) Now that we've got a working example of movable cards laid out nicely on screen, I begin to think about how the game will be played (step 4 in the overall program outline). The first thing I realize is that there's absolutely nothing in the "movestyle" code that allows me to determine which card I'm touching at any given moment. Those cards are simply collections of binary graphic data displayed on the screen. The only unique and meaningful information I can get about any given card image is its current position (face/offset). To operate the game, however, I need to know a card's face value, suit, color, etc. A simple solution to that problem can be managed by keeping track of each card's current position, and mapping those locations to individual card face values. To do that, I'll create a block that maps each specific card to its initial coordinates, and then update that block every time a move is made. Every time a card is moved, the given card's new coordinates will overwrite the old coordinates in the block. That way, I can figure out a card's face value simply by reversing the thought process. By looking up any card's

specific current position, I can perform conditional evaluations on the related face value, suit, color, etc. of the card at that location ... Sneaky, huh? Doing that additionally allows me to keep track of how the piles of cards are laid out in the playing field - that's also going to play an important role in the game.

The block that maps coordinates to face values can be easily created during the foreach loop that builds the gui layout. In order to keep track of the positions of each card, I need to define some additional variables. Because I used REBOL's automatic GUI layout capabilities to arrange the cards on screen, rather than specifically positioning each image, there are currently no existing variables that store any of those position values. To store the starting coordinates of each card, I create the additional variables "cardnumber" and "xpos", along with a block labled "card-coords" to hold all the values. Now, as each card's image data is layed out in the gui block, it's position is calculated and appended to the card-coords block:

```
gui: [across ]
card-coords: copy []
coord: 0x0
count: 0
cardnumber: 1
ypos: 20
xpos: 20
foreach card cards [
    if count = 8 [
        ypos: ypos + 30   ; start a new row every 8 cards
        xpos: 20
        coord: to-pair rejoin [20 "x" to-string compose (ypos)]
        append gui compose [at (coord)]
        count: 0
    ]
    append gui compose [
        image load to-binary decompress (card) feel movestyle
    ]
    coord: to-pair rejoin [to-string compose (xpos)
        "x" to-string compose (ypos)]
    print coord
    ; Add the coordinate of the newly created card to the
    ; card-coords block:
    append card-coords compose [(cardnumber) (coord)]
    count: count + 1
    cardnumber: cardnumber + 1
    xpos: xpos + 79
]
view layout gui

print "The cards and their positions are: "
probe card-coords halt
```

Tada! Now every card has a numeric label and each label is tied to a specific starting position.

```
; These numbers are significant, because all the cards will be
; referred to by them throughout the rest of the game.

1-13  = clubs         ace through king
14-26 = diamonds      ace through king
27-39 = hearts        ace through king
40-52 = spades        ace through king
```

It's important to note that all changes to the current coordinates, and any other calculations, conditional evaluations, etc. need to occur within the "movestyle" block. Remember, the gui block is simply statically created - it's only run once when the script is first evaluated from beginning to end.

It's the movestyle block of code that gets evaluated every time a card is touched. We can include any code that needs to be run, inside the sections that are evaluated when either a down, over, or away action is detected. For example, we can use the following formula to find the index number of the card's initial coordinate in the card-coords block:

```
new-pos: (index? find card-coords start-coord) / 2
; divide the index number by two because each card has two values:
; position number in the grid, and coordinate.
```

With that index number, we can look up the name of the card at the same index position in a new "card-names" block using the code "card-names/:new-pos" (that reads: the item in the card-names block at the index number created above). In this example, those values are calculated inside the movestyle block, and a message is printed to demonstrate the mapping technique. Take a close look at the print code to see how each of the variables work:

```
card-names: [
    "ace of clubs" "2 of clubs" "3 of clubs" "4 of clubs"
    "5 of clubs" "6 of clubs" "7 of clubs" "8 of clubs" "9 of clubs"
    "10 of clubs" "jack of clubs" "queen of clubs" "king of clubs"
    "ace of diamonds" "2 of diamonds" "3 of diamonds"
    "4 of diamonds"    "5 of diamonds" "6 of diamonds" "7 of diamonds"
    "8 of diamonds" "9 of diamonds" "10 of diamonds"
    "jack of diamonds" "queen of diamonds" "king of diamonds"
    "ace of hearts" "2 of hearts" "3 of hearts" "4 of hearts"
    "5 of hearts" "6 of hearts" "7 of hearts" "8 of hearts"
    "9 of hearts" "10 of hearts" "jack of hearts" "queen of hearts"
    "king of hearts" "ace of spades" "2 of spades" "3 of spades"
    "4 of spades" "5 of spades" "6 of spades" "7 of spades"
    "8 of spades" "9 of spades"    "10 of spades" "jack of spades"
    "queen of spades" "king of spades"
]

movestyle: [
    engage: func [face action event] [
        if action = 'down [
            start-coord: face/offset
            if (find card-coords start-coord) [
                new-pos: (index? find card-coords start-coord) / 2
                print rejoin [
                    "You touched card number: "
                    (select card-coords new-pos)
                    "^/Position #"
                    new-pos " in the grid."
                    newline
                    card-names/:new-pos
                    newline
                ]
            ]
            face/data: event/offset
            ; remove find face/parent-face/pane face
            ; append face/parent-face/pane face
        ]
        if find [over away] action [
            unrounded-pos: (face/offset + event/offset - face/data)
            snap-to-x: (round/to first unrounded-pos 79) + 20
            snap-to-y: (round/to second unrounded-pos 30) + 20
            face/offset: to-pair rejoin [snap-to-x "x" snap-to-y]
            print face/offset
        ]
        show face
```

```
    ]
]

gui: [across ]
card-coords: copy []
coord: 0x0
count: 0
cardnumber: 1
ypos: 20
xpos: 20
foreach card cards [
    if count = 8 [
        ypos: ypos + 30
        xpos: 20
        coord: to-pair rejoin [20 "x" to-string compose (ypos)]
        append gui compose [at (coord)]
        count: 0
    ]
    append gui compose [
        image load to-binary decompress (card) feel movestyle
    ]
    coord: to-pair rejoin [to-string compose (xpos) "x"
        to-string compose (ypos)]
    print coord
    append card-coords compose [(cardnumber) (coord)]
    count: count + 1
    cardnumber: cardnumber + 1
    xpos: xpos + 79
]
view layout gui
```

Using those techniques, I can finish up the card tracking code. The following code makes changes to the card-coords block every time a card is moved:

```
movestyle: [
    engage: func [face action event] [
        if action = 'down [
            start-coord: face/offset
            print find card-coords start-coord
            face/data: event/offset
            remove find face/parent-face/pane face
            append face/parent-face/pane face
        ]
        if find [over away] action [
            unrounded-pos: (face/offset + event/offset - face/data)
            snap-to-x: (round/to first unrounded-pos 79) + 20
            snap-to-y: (round/to second unrounded-pos 30) + 20
            face/offset: to-pair rejoin [snap-to-x "x" snap-to-y]
        ]
        show face
    ]
]
```

And with that, we have the necessary code to keep track of every card's position. The program as it currently exists is a useful generic foundation for any card game. Now we need to begin working on the game logic for Freecell. Here are the main objectives, along with some pseudo-code ideas to help organize the thought process:

1. If a black card is placed below any red card at the bottom of any of the 8 "physical" piles of cards, or visa-versa (red-black), check to see if the moved card is 1 card lower in value than the card it touches. If not, don't allow the move. For example, a red 8 can be moved below a black

9, but moving a red 8 below a red 9 (not alternate red-black), or a black king beneath a red 3 (not consecutive), isn't allowed. You can make a disallowed card movement happen programmatically by resetting the face/offset of any disallowed card back to the value it held before being moved (that value must therefore be saved at soon as a card is touched).
2. Only cards exposed at the bottom of pile, or one of the cards in a descendingly stacked group of alternate red-black cards at the bottom of a pile can be moved. For example, in a group of cards r7, b6, r5, b4, r3 at the bottom of a pile, you can move the red 7 and all the cards underneath it to another pile with an exposed black 8 at the bottom of the pile. You could also grab the black 4 and move it, along with the red 3 together, beneath a pile with a red 5 at the bottom. You could not, however, grab the red 7 from that pile without also moving the rest of the cards (b6, r5, b4, r3) beneath it.
3. The goal of the game is to move all cards from the originally displayed 8 piles to 4 new "goal" piles that are initially empty. Upon completion, each pile must contain only cards of a unique suit (clubs, diamonds, hearts, or spades) and the face values must ascend from ace to king consecutively. Disallow any card movements that don't allow for that arrangement.
4. There are 4 additional spaces, or "free cells" (the name of the game), that can be used to temporarily hold and move cards around between piles. They are useful in moving cards when there are no positions within the initial piles or in the goal piles that allow a card to be moved according to the previous rules. Only single exposed cards (no covered cards or piles) can be moved to a free cell.

With all that done, I've noticed a little bug in the previous example. When I move a card, it sits on top of, and covers the other cards. That's only desirable if the moved card is the bottom card in it's pile. I'll solve that problem by using the existing block of card coordinates. Every time a card is moved, I'll check to see if it's moved to the lowest position in each pile. If so, I'll make it sit on top of the other cards. If not, it will retain a position underneath the other cards.

The next problem comes when I want to move an entire pile of cards.

Next, I need to randomize the original position of each card. The easiest way that I can think to do that is to create a loop that moves each card around the screen randomly, as if they'd been moved by hand. Just run a feel movestyle on each card.

Next, I'll create a simple background to frame the playing field.

10.21 Case 21 - An Additional Teacher Automation Project

Now that the group scheduling system is complete, I want to automate our daily checkout routine. Every day, our teachers are paid directly by their students. In turn, they pay us a rental/referral fee for room and resource use. That's our primary source of income. At the end of the day, teachers add up all the students they've seen, and pay a given fee for each completed half hour session. Some students prepay their teachers, and the teachers in turn prepay us so that they don't have to manage rental fees for prepaid appointments in the future.

It takes a lot of time to manually figure daily fees, and the process is error prone when calculated by hand. I want to automate the payment calculations based on the existing online schedule information, and I want to create an integrated record keeping system to more easily track prepayments. Teachers need to keep track of missed/rescheduled appointment payments, so that students are given proper credit for rolled over appointment times. Also, in addition to daily local lessons, some of our instructors teach online lessons, for which _we're_ paid directly by students. For those lessons, we deduct room rent from the total paid to us by the teachers. We need a solution to easily manage and track all those daily calculations for all the teachers. The objective is to keep a running total of how much money is due by each teacher every night, and how much money is owed to the teachers by students. To create a software outline, I thought about what I do manually every day to calculate the checkout fees for a single person. This thought process will serve as an outline to design the automated record keeping system:

1. Each day at checkout time, the total number of lessons for a teacher is added up.
2. The teacher owes us a given amount for lessons that occurred in the local studio that day.
3. We owe the teacher a given deduction for each lesson they performed online that day.
4. Any lessons which had previously been prepaid by the teacher are deducted from the total owed us.
5. The teacher prepays us for any future lessons which were prepaid by students that day, and records are updated to track the current prepaid amounts.
6. Occasionally, other deductions are made from the amount owed us (sometimes the teacher provides a complimentary lesson for various reasons, or we provide complimentary time to the teacher/student, etc.). Those amounts are deducted from the total owed us.

Based on the guidelines above, here's how I organized my thoughts about what the automated multiuser system should do:

1. The multiuser requirements of the application are similar to those of the scheduling app from the previous section. I can use the code from the scheduling app to provide a current teacher list, simple password protection, loading/saving/backup of required data files for the selected teacher, etc.
2. In order to perform daily calculations for a single instructor, I want to provide a dynamically created list of daily students and I want to retrieve current prepay records for the given teacher. That data will be stored on the web site, any changes will be backed up locally and on the web site. I'll need to come up with a data structure to store the prepay records. All other information (random deductions, complimentary lessons, etc.) will be provided by the user on a daily basis. The regular daily student list and prepay records can be downloaded and displayed in text lists. The other random deductions and additions can be entered manually in text input fields, and displayed in text lists.
3. By default, each teacher owes a given amount for each of the students selected from the daily list (number_of_students X half_hour_rate). Add to that any fees for additional students not in the daily list (rescheduled lessons, occasional additional appointments, etc.)
4. For each online lesson, subtract 1 student from the total number of students taught that day, and deduct the appropriate amount from the grand total due.
5. Subtract any previous prepayments from the grand total due. Whenever that happens, make an adjustment to the teacher's record of prepayments.

To satisfy step 1, I'll use the scheduling app from the previous section of this case study. As it stands, that code is capable of selecting a specified teacher directory on the website and downloading any required data files (current daily students, prepay records, individual teacher fee rates, etc.).

The main work of creating the application is in step 2. The required calculations are in steps 3-5. Here's a more structured outline, with pseudo code, to guide the writing of the program code:

1. Read a current list of daily students from the selected teacher's schedule.txt file on the web server. Store that info in a block and display it in a GUI text-list widget. Store a list of local students selected from the above widget in a separate block.
2. Display today's students again in a second text-list widget, so that the user can select those who took lessons online. Store that selected data in another block.
3. Provide a text input field to allow the addition of any students not in the daily list. Display a text-list widget to contain students entered into the text field. Update the text-list display any time a student is added. In order to remove incorrect entries from this list, the action block of the text-list should contain code to delete any students selected by the user.
4. Provide another text field and text-list for the entry of deductions, with the same layout and remove code.
5. Provide a button to manage prepayment entries and calculations. To handle that whole process, create a separate script - to be outlined below.
6. Provide a "Calculate Total Fees" button. The action block of this button should add and subtract the total number of items in all of the text-lists, according to the rules defined in steps 3-5 of the overall program outlined above. Provide an HTML summary, which the teacher can print out and submit every day.

Here's the code I came up with to do all that:

```
; The "url" variable below comes from the multiuser framework
; borrowed from the scheduler app:
students: read/lines rejoin [url "/schedule.txt"]
; Initialize some other variables:
other-additions: []    other-deductions: [] prepays: []
pay-for: copy [] online: copy []

view center-face layout [
    h2 "Local Students:"
    ; "face/picked" refers to the currently selected items in
    ; the text-list (use [Ctrl] + mouse click to select multiple
    ; items, and assign that to the variable "pay-for":
    text-list data copy students [pay-for: copy face/picked]
    h2 "Other Additions:"
    field [
        ; add the entered info to the text-list, and update
        ; the display:
        append other-additions copy face/text
        show other-additions-list
    ]
    other-additions-list: text-list 200x100 data other-additions [
        ; remove any entry when selected by the user, and update
        ; the display
        remove-each item other-additions [item = value]
        show other-additions-list
    ]
    at 250x20
    h2 "Online Students:"
    text-list data copy students [online: face/picked]
    h2 "Other Deductions:"
    field [
        append other-deductions copy face/text
        show other-deductions-list
    ]
    other-deductions-list: text-list 200x100 data other-deductions [
        remove-each item other-deductions [item = value]
        show other-deductions-list
    ]
    at 480x20
```

```
    h2 "Prepaid Lessons:"
    prepay-list: text-list data prepays [
        remove-each item prepays [item = value]
        show prepay-list
    ]
    ; I still need to create the prepay.r program:
    btn 200x30 "Calculate Prepaid Lessons" [
        save %prepay.txt load rejoin [url "/prepay.txt"]
        do %prepay.r
    ]
    at 480x320
    btn 200x100 font-size 17 "Calculate Total Fees" [
        total-students: (
            (length? pay-for) - (length? online) +
            (length? other-additions) - (length? other-deductions) -
            (length? prepays)
        )
        ; I want to create an HTML output for this section:
        alert rejoin ["Total: " to-string total-students]
    ]
]
```

Now all that's left to create is a separate program to manage prepayment info. Here's an outline describing my intentions:

1. Create and upload a "prepay.txt" data file to store prepayment information for each teacher. It should contain a separate block for each student who prepays, with fields for the student name, a nested block for the amounts and dates of each prepayment, and a nested block for dates of each lesson attended and the amount deducted from the prepayment for each lesson.
2. Create a GUI with a text-list displaying each student who has prepayed. Loop through the prepay.txt data to get the student names (the first item in each block). Whenever a name is selected by the user, display the student name, prepay dates and amounts, and lesson dates in separate text lists. Display the total prepay balance for the selected student in a text field.
3. There should be an "Add" button and some text fields for entering new prepayments. There should be fields for student name, amount, and date of prepay. If an existing student is selected from the list, those fields should be populated automatically with today's date, and with the name of the existing student. The action block of the add button should append the information to appropriate blocks in the prepay.txt file.
4. There should be an "Apply Today" button to select prepayment(s) to be applied to today's balance. Store the names of the selected students in a block, save that block to be read and used in the main application, and add the date information to the appropriate blocks in the prepay.txt file.
5. There should be an "Done" button on the list-view GUI to allow the information to be changed and saved. Whenever a student is selected from the list, their prepayment records should be displayed in an editable list-view (import the listview module and use the database example from earlier in this tutorial as a model). There should be fields for prepay amounts and dates, and lesson dates and amounts.
6. When the main prepay application is closed, the prepay.txt file should be backed up and saved to the web site.

For step 1, here's an example of the block structure I came up with to store data in the prepay.txt file:

```
[
    ; name:
    "John Smith"

    ; prepayment amounts and dates:
    [ [$100 4-April-2006] [$100 5-May-06] ]

    ; dates of lessons:
```

```
    [
        [$20 4-April-06] [$20 11-April-06] [$20 18-April-06]
        [$20 25-April-06] [$20 5-May-06]
    ]
]

[
    "Paul Brown"

    [ [$100 4-April-2006] ]

    [
        [$20 4-April-06] [$20 25-April-06]
    ]
]

[
    "Bill Thompson"

    [ [$200 22-March-2006] ]

    [
        [$20 22-March-06] [$20 29-March-06] [$20 5-April-06]
        [$20 12-April-06] [$20 19-April-06]    [$20 26-April-06]
        [$20 3-May-06]
    ]
]


[
    ; name:
    "John Smith"

    ; prepayment amounts and dates:
    [ "$100 4-April-2006" "$100 5-May-06" ]

    ; dates of lessons:
    [
        "$20 4-April-06" "$20 11-April-06" "$20 18-April-06"
        "$20 25-April-06" "$20 5-May-06"
    ]
]

[
    "Paul Brown"

    [ "$100 4-April-2006" ]

    [
        "$20 4-April-06" "$20 25-April-06"
    ]
]

[
    "Bill Thompson"

    [ "$200 22-March-2006" ]

    [
        "$20 22-March-06" "$20 29-March-06" "$20 5-April-06"
        "$20 12-April-06" "$20 19-April-06" "$20 26-April-06"
        "$20 3-May-06"
    ]
```

```
]
```

Here's the code I created to fulfill my outline requirements:

```
REBOL [title: "Prepayment Calculator"]

prepays: load rejoin [url "/prepay.txt"]
names: copy []
prepay-history: []
lesson-history: []
display-todays-bal: does [
    ; calculate and display the current balance for the
    ; selected student:
    todays-balance: $0
    foreach payment prepay-history [
        todays-balance: todays-balance + (
            first (to-block payment)
        )
    ]
    foreach lesson-event lesson-history [
        todays-balance: todays-balance - (
            first (to-block lesson-event)
        )
    ]
    ; update the display of today's balance for the
    ; selected student :
    today-bal/text: to-string todays-balance
    show today-bal
]
foreach block prepays [append names first block]
view center-face gui: layout [
    across
    text bold "New Prepayment:"
    text right "Name:" new-name: field
    text right "Date:" new-date: field 125 to-string now/date
    text right "Amount:" new-amount: field 75 "$"
    btn "Add" [
        create-new-block: true
        foreach block prepays [
            if (first block) = new-name/text [
                create-new-block: false
                append (second block) to-string rejoin [
                    new-amount/text " " new-date/text
                ]
            ]
        ]
        if create-new-block = true [
            new-prepay: copy []
            append new-prepay to-string new-name/text
            append new-prepay to-string rejoin [
                new-amount/text " " new-date/text
            ]
            append prepays new-prepay
            names: copy []
            foreach block prepays [append names first block]
        ]
        display-todays-bal
        show existing show pre-his show les-his show today-bal
    ]
    return
    text bold underline "Edit Data Manually" [
```

```
        view/new center-face layout [
            new-prepays: area 500x300 mold prepays
            btn "Save Changes" [
                prepays: copy new-prepays/text
                unview
            ]
        ]
        names: copy []
        foreach block prepays [append names first block]
        show gui
        show existing show pre-his show les-his show today-bal
    ]
    return
    text "Existing Prepayments:"  pad 75
    text "Prepayment History:"  pad 85
    text "Lesson History:" pad 100
    text "Balance:"
    return
    existing: text-list data names [
        ; When a name is selected from this text list, update
        ; the other fields on the screen:
        new-name/text: value
        show new-name
        foreach block prepays [
            if (first block) = value [
                ; update the other text lists to show the
                ; selected student's prepay and lesson history:
                prepay-history: pre-his/data: second block
                show pre-his
                lesson-history: les-his/data: third block
                show les-his
            ]
        ]
        display-todays-bal
        ; get the list of selected students
        prepaid-today: copy face/picked
    ]
    pre-his: text-list data prepay-history
    les-his: text-list data lesson-history
    today-bal: field 85
    return
    btn "Apply Selected Prepayments Today" [
        save %prepaid.txt prepaid-today

        unview
    ]
]
```

In the original scheduling outline, I replace all references in the code to "schedule.txt" with "prepay.txt":

```
REBOL [title: "Payment Calculator"]

error-message: does [
    ans: request {Internet connection is not available.
        Would you like to see one of the recent local backups?}
    either ans = true [
        editor to-file request-file quit
    ][
        quit
    ]
```

```
]

if error? try [
    teacherlist: load ftp://user:pass@website.com/teacherlist.txt
][
    error-message
]
teachers: copy []
foreach teacher teacherlist [append teachers first teacher]
view center-face layout [
    text-list data teachers [folder: value unview]
]

pass: request-pass/only
correct: false
foreach teacher teacherlist [
    if ((first teacher) = folder) and (pass = (second teacher)) [
        correct: true
    ]
]
if correct = false [alert "Incorrect password." quit]

url: rejoin [http://website.com/teacher/ folder]
ftp-url: rejoin [
    ftp://user:pass@website.com/public_html/teacher/ folder
]

if error? try [
    write %prepay.txt read rejoin [url "/prepay.txt"]
][
    error-message
]

; backup (before changes are made):
cur-time: to-string replace/all to-string now/time ":" "-"
; local:
write to-file rejoin [
    folder "-prepay_" now/date "_" cur-time ".txt"
] read %prepay.txt
; online:
if error? try [
    write rejoin [
        ftp-url "/" now/date "_" cur-time
    ] read %prepay.txt
][
    error-message
]

editor %prepay.txt

; backup again (after changes are made):
cur-time: to-string replace/all to-string now/time ":" "-"
write to-file rejoin [
    folder "-prepay_" now/date "_" cur-time ".txt"
] read %prepay.txt
if error? try [
    write rejoin [
        ftp-url "/" now/date "_" cur-time
    ] read %prepay.txt
][
    alert "Internet connection not available while backing up."
]
```

```
; save to web site:
if error? try [
    write rejoin [ftp-url "/prepay.txt"] read %prepay.txt
][
    alert {Internet connection not available while updating web
    site.  Your schedule has NOT been saved online.}
    quit
]
browse url
```

I also need to replace the line "editor %prepay.txt" with new code that does the work of calculating daily fees and tracking prepayments.

Now that the program is complete, please notice how the outline developed. It took several steps. First, I thought through my daily manual calculations. Then I thought about how that could be encapsulated into a program, and I created a basic outline about what I wanted the program to do. When it came to writing pseudo code outlines to create the actual program, the whole process was made easier by having organized outlines of everything I needed to accomplish. To write the program, I first defined some required data (provided by the multiuser scheduler app), then conceived a user interface, and then performed calculations based on existing data and user input. Following that type of outline structure (define required data, define a UI, perform calculations) tends to be an organized and successful approach in many cases.

It should be noted that I'm not concerned about data security in this app. It is important that the teachers are able to access this info conveniently from any location. It's also important that local backups are made. The automatic backing up of files provides a historical audit trail of transactions and changes to the records, which is an important concern since this program manages income. It's not a problem for these records to become publicly accessible, so I'm using ftp and a public web site to store and retrieve the data. Writing secure applications, however, is an important requirement in most situations involving financial transactions. You should be aware that data security is a primary concern if you intend to do any programming related to typical business transactions, but that topic is beyond the scope of this tutorial. This case study was provided as an additional example of how coding thought can be organized to take you from conceptual phases to a final product. This particular code should not be emulated, however, for projects requiring secure data transactions.

11. Other Scripts

This section of the tutorial contains various random scripts that you might find useful.

The first script provides a quick visual reference of all REBOL's built in colors:

```
REBOL [Title: "Quick Color Guide"]

echo %colors.txt ? tuple! echo off
lines: read/lines %colors.txt
colors: copy []
gui: copy [across space 1x1]
count: 0
foreach line at lines 2 [
    if error? try [append colors to-word first parse line none][]
]
foreach color colors [
    append gui [style box box [alert to-string face/color]]
    append gui reduce ['box 110x25 color to-string color]
    count: count + 1
    if count = 5 [append gui 'return count: 0]
]
view center-face layout gui
```

This next quick script demonstrates how to convert REBOL color tuples to HTML colors, and visa-versa:

```
to-binary request-color
to-tuple #{00CD00}
view layout [box to-tuple #{5C743D}]  ; view an HTML color on screen!
```

This is a quick way to review the console history of your current REBOL session:

```
foreach line reverse copy system/console/history [print line]
```

The script below demonstrates how to use email ports to read one message at a time from a pop server. Be sure to set your email user account settings before running this one (that's explained earlier in this tutorial):

```
for i 1 length? pp: open pop://user@site.com 1 compose [
    ask find pp/(i) "Subject:"
]
```

Here are a few examples of how the "request" function can be used:

```
; Two different formats include passing either a string or a block.
; If you pass a string, the default buttons will be "yes", "no", and
; "cancel".  If you pass a block, you can specify the text on the
; buttons:

request "Could this be useful?"
request ["Just some information."]
request ["Here are 2 buttons with altered text:" "Probably" "Not Really"]
request ["3 buttons with altered text:" "Probably" "Not Really" "Dunno"]
```

```
; "Request" only returns 'true 'false or 'none.  For an example like the
; one below, the user response can be converted to strings using a switch
; structure:

answer: form request ["Complex example:" "choice 1" "choice 2" "choice 3"]
switch/default answer [
    "true"  [the-answer: "choice 1"]
    "false" [the-answer: "choice 2"]
    "none"  [the-answer: "choice 3"]
] []
print the-answer

; The "/type" modifier changes the icon displayed:

request/type ["Here's a better icon for information display."] 'info
request/type ["Altered title and button text go in a block:" "Good"] 'info
request/ok/type "This example is the EXACT same thing as 'alert'." 'alert
request/ok/type "Here's alert with a different icon." 'info
request/ok/type "Here's another icon!" 'stop
halt
```

Here is a home made resizable requestor that I use when I don't want the "REBOL - " title bar to appear. It has a default timeout (set to 6 seconds in the example below), and can also be closed with a button click. This is particularly suitable for full screen commercial kiosk types of applications):

```
sz: 5
view layout [
    btn "Click here to see a requestor with a 6 second timeout" [
        view/new/options center-face information: layout [
            text font-size (8 * sz) "Here's a message!" rate :00:06 feel [
                engage: func [f a e] [
                    if a = 'time [unview/only information]
                ]
            ]
            box 1x1  ; spacer
            btn as-pair (12 * sz) (8 * sz) font-size (5 * sz) "Ok" [
                unview/only information
            ]
        ][no-title]
    ]
]
```

To edit the source of any mezzanine function, use the following format:

```
editor mold :request
editor mold :inform
```

I actually use the following script to check the source files of this tutorial, to make sure that none of the lines of code are wider than 79 characters:

```
REBOL [title: "Find Long Lines"]

doc: read/lines to-file request-file
the-text: {}
foreach line doc [
    if ((find/part line "   " 4)) [
        if ((length? line) > 78) [
```

```
            print line
            the-text: rejoin [the-text newline line]
        ]
    ]
]
editor the-text
```

Here's a duo of scripts that I use to sync my computer's clock to the time and date on my web server. The Windows API time setting function is based on Ladislav Mecir's Nist Clock Sync Script:

```
REBOL []

dif: 7:00  ; difference between web server and your local time zone
date: (to-date trim read http://site.com/time.cgi) + dif

lib: load/library %kernel32.dll

set-clock: make routine! [
    systemtime [struct! []]
    return:    [integer!]
] lib "SetSystemTime"

current: make struct! [
    wYear         [short]
    wMonth        [short]
    wDayOfWeek    [short]
    wDay          [short]
    wHour         [short]
    wMinute       [short]
    wSecond       [short]
    wMilliseconds [short]
] reduce [
    date/year
    date/month
    date/weekday
    date/day
    date/time/hour
    date/time/minute
    to-integer date/time/second
    0
]

either ((set-clock current) = 1) [
    ask rejoin ["Time has been set to:  " now "^/^/[Enter]... "]
] [
    ask "Error setting time.  Please check your Internet connection."
]

free lib
```

Here's the CGI script that the above code needs (to obtain the date and time from the web server). Put it at the URL which is read when the 'date word above is set:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL [title: "time"]
print "content-type: text/html^/"
print now
```

Here's Ladislav's (better) version of the above Windows function. The script at http://www.fm.tul.cz

/~ladislav/rebol/nistclock.r can set both Linux *and* Windows system clocks (the "set-system-time-lin" does the same thing in Linux):

```
the-date: to-date trim read http://site.com/time.cgi

set-system-time-win: func [
    {set system time in Windows; return True in case of success}
    [catch]
    date
    /local set-system-time
] [
    unless value? 'kernel32 [kernel32: load/library %kernel32.dll]
    set-system-time: make routine! [
        systemtime [struct! []]
        return: [int]
    ] kernel32 "SetSystemTime"
    date: date - date/zone
    date/zone: 0:0
    0 <> set-system-time make struct! [
        wYear [short]
        wMonth [short]
        wDayOfWeek [short]
        wDay [short]
        wHour [short]
        wMinute [short]
        wSecond [short]
        wMilliseconds [short]
    ] reduce [
        date/year
        date/month
        date/weekday
        date/day
        date/time/hour
        date/time/minute
        to integer! date/time/second
        0
    ]
]

set-system-time-win the-date
```

I use the following script to upload screen shots of my desktop directly to my web site (in the version I use, I put the text of the included script directly into my code):

```
REBOL []

do http://www.rebol.org/download-a-script.r?script-name=capture-screen.r

the-image: ftp://user:pass@site.com/path/current.png

; You can also save to your local hard drive if you want:
; the-image: %current.png

view center-face gui: layout [
    button 150 "Upload Screen Shot" [
        unview gui
        wait .2
        save/png the-image capture-screen
        view center-face gui
    ]
```

```
]
```

The following script demonstrates how to add and remove widgets from a GUI layout:

```
view gui: layout  [
    button1: button
    button2: button "remove" [
        remove find gui/pane button1
        show gui
    ]
    button3: button "add" [
        append gui/pane button1
        show gui
    ]
]
```

Here's a way to get a unique string identifier from the current time (useful for MCI buffer names and other situations when you need to generate absolutely unique identifier strings without any odd characters):

```
replace/all replace/all replace/all form now "/" "" ":" "x" "-" "q" "." ""

; precise version (w/ milliseconds):

replace/all replace/all replace/all replace/all form now/precise trim {
    /} "" ":" "x" "-" "q" "." ""
```

This script creates an image, saves it to the hard drive, and then opens it in mspaint.exe:

```
save/bmp %test.bmp to-image layout [box]
call/show join "mspaint.exe " to-local-file join what-dir %test.bmp
```

This script demonstrates how to use the AutoIT DLL to control the madplay.exe mp3 player:

```
REBOL []

if not exists? %AutoItDLL.dll [
    write/binary %AutoItDLL.dll
    read/binary http://musiclessonz.com/rebol_tutorial/AutoItDLL.dll
    write/binary %madplay.exe
    read/binary http://musiclessonz.com/rebol_tutorial/madplay.exe
]

lib: load/library %AutoItDLL.dll

move-mouse: make routine! [
    return: [integer!] x [integer!] y [integer!] z [integer!]
] lib "AUTOIT_MouseMove"

send-keys: make routine! [
    return: [integer!] keys [string!]
] lib "AUTOIT_Send"

winactivate: make routine! [
    return: [integer!] wintitle [string!] wintext [string!]
] lib "AUTOIT_WinActivate"
```

```
set-option: make routine! [
    return: [integer!] option [string!] param [integer!]
] lib "AUTOIT_SetTitleMatchMode"

set-option "WinTitleMatchMode" 2
call/show {madplay.exe -v *.mp3}

view layout [
    across
    btn "forward" [
        winactivate "\reb" ""
        send-keys "f"
    ]
    btn "back"[
        winactivate "\reb" ""
        send-keys "b"
    ]
    btn "volume up" [
        winactivate "\reb" ""
        send-keys "+"
    ]
    btn "volume-down"[
        winactivate "\reb" ""
        send-keys "-"
    ]
    btn "pause" [
        winactivate "\reb" ""
        send-keys "p"
    ]
    btn "quit" [
        winactivate "\reb" ""
        send-keys "q"
        quit
    ]
]
```

Here's a quick and dirty way to print out help for all built in functions. Also includes a complete list of VID styles ("view layout" GUI widgets), VID layout words, and VID facets (standard properties available for all the VID styles). Give it a minute to run...

```
REBOL [title: "Quick Manual"]

print "This will take a minute..."  wait 2
echo %words.txt what echo off   ; "echo" saves console activity to a file
echo %help.txt
foreach line read/lines %words.txt [
    word: first to-block line
    print "_____________________________________________________________^/"
    print rejoin ["word:  " uppercase to-string word]  print ""
    do compose [help (to-word word)]
]
echo off
x: read %help.txt
write %help.txt "VID STYLES (GUI WIDGETS):^/^/"
foreach i extract svv/vid-styles 2 [write/append %help.txt join i newline]
write/append %help.txt "^/^/LAYOUT WORDS:^/^/"
foreach i svv/vid-words [write/append %help.txt join i newline]
b: copy []
foreach i svv/facet-words [
    if (not function? :i) [append b join to-string i "^/"]
```

```
]
write/append %help.txt rejoin [
    "^/^/STYLE FACETS (ATTRIBUTES):^/^/" b "^/^/SPECIAL STYLE FACETS:^/^/"
]
y: copy ""
foreach i (extract svv/vid-styles 2) [
    z: select svv/vid-styles i
    ; additional facets are held in a "words" block:
    if z/words [
        append y join i ": "
        foreach q z/words [if not (function? :q) [append y join q " "]]
        append y newline
    ]
]
write/append %help.txt rejoin [
    y "^/^/CORE FUNCTIONS:^/^/" at x 4
]
editor %help.txt
```

Here's an email program that demonstrates how to set all your email account settings:

```
m: system/schemes/default q: system/schemes/pop
view layout [ style f field
    u: f "username" p: f "password" s: f "smtp.address" o: f "pop.address"
    btn bold "Save Server Settings" [
        m/user: u/text m/pass: p/text m/host: s/text q/host: o/text
    ] tab
    e: f "user@website.com" j: f "Subject" t: area
    btn bold "SEND" [
        send/subject to-email e/text t/text j/text alert "Sent"
    ] tab
    y: f "your.email@somesite.com"
    btn bold "READ" [foreach i read to-url join "pop://" y/text [ask i]]
]
```

This example keeps a real time word count of text in an area widget. Changing the rate will reduce system resource usage, but also slow the response time (a rate of 3-4 updates per second should be suitable for most cases):

```
view layout [
    i: info rate 0 feel [
        engage: func [f a e] [
            if a = 'time [
                l: length? parse m/text none
                i/text: join "Wordcount: " l
                show i
            ]
        ]
    ]
    m: area
]
```

Here are two versions of the VOIP program given earlier in the section about ports. These are likely the most compact VOIP programs you'll find anywhere. The first features port error handling, automatic localhost testing (just press [ENTER] to use localhost as the IP address), hands-free operation, and automatic minimum volume testing (squelch - data not sent unless a given volume is detected, to save bandwidth). It can be pasted directly into the REBOL console, or saved to a file and run. The second is barebones (user sees errors when the connection is broken, must be saved to a file and run, etc.) but it does work. The file sizes of these scripts are 693 bytes and 554 bytes!!:

```
REBOL[]do[write %w{REBOL[]if error? try[p: first wait open/binary/no-wait
tcp://:8][quit]wait 0 s: open sound:// forever[if error? try[m: find v:
copy wait p #""][quit]i: to-integer to-string copy/part v m while[i >
length? remove/part v next m][append v p]insert s load to-binary
decompress v]}launch %w lib: load/library %winmm.dll x: make routine![c[
string!]return:[logic!]]lib"mciExecute"if(i: ask"Connect to IP: ")=""[i:
"localhost"]if error? try[p: open/binary/no-wait rejoin[tcp:// i":8"]][
quit]x"open new type waveaudio alias b"forever[x"record b"wait 2 x
"save b r"x"delete b from 0"insert v: compress to-string read/binary
%r join l: length? v #""if l > 4000[insert p v]]]

REBOL[]write %w{REBOL[]if error? try[p: first wait open/binary/no-wait
tcp://:8][quit]wait 0 s: open sound:// forever[m: find v: copy wait p #""
i: to-integer to-string copy/part v m while[i > length? remove/part v next
m][append v p]insert s load v]}launch %w lib: load/library %winmm.dll x:
make routine![c[string!]return:[logic!]]lib"mciExecute"i: ask"IP: "p:
open/binary/no-wait rejoin[tcp:// i":8"]x"open new type waveaudio alias b"
forever[x"record b"wait 2 x"save b r"x"delete b from 0"insert v:
read/binary %r join length? v #""insert p v]
```

Here's an instant message example that allows users to upload their connection info (username, WAN IP, LAN IP, and network port), to a text file on an FTP server. Then others can simply click on their user name in a drop down box (choice button), to connect:

```
REBOL [title: "Instant Messenger"]

servers: ftp://username:password@yoursite.com/public_html/im.txt  ; EDIT
flash "Retrieving server list..."
if error? try [server-info: reverse read/lines servers] [
    alert "Internet connection not available."
    server-info: copy []
]
unview
name-list: copy []
foreach server server-info [append name-list first to-block server]
insert head name-list "Connect to a Server:"

view center-face layout [
    across
    choice 280 data name-list [
        mode: request [ {
            SELECT MODE: By default, this program is able to connect to
            a server running on any other computer in your Local Area
            Network.  Choosing "LAN" mode connects you to a server's local
            IP address.  If you select "Internet" Mode, you will connect
            to the server's WAN IP address (typically the address of
            the user's _router_).  In order for Internet mode to work
            correctly, the selected port number chosen by the server user
            must be exposed on the Internet, or be forwarded from their
            router to the IP address of the computer running the server
            program.
        } "LAN" "Internet"]
        foreach server server-info [
            server-block: parse server " "
            if ((form first server-block) = form value) [
                b/text: server-block/1  show b
                either mode = false [
                    remote-ip: server-block/2
                ] [
                    remote-ip: server-block/3
```

```
                ]
                j/text: server-block/4
                show j
                p: open/lines rejoin [tcp:// remote-ip j/text]
                z: 1
                focus g
                break
            ]
        ]
    ]
    text "OR run as server:"
    b: field 106 "Username"
    text "Port: "
    j: field 55 ":8080"
    q: button 84 "Start Server" [
        parse read http://guitarz.org/ip.cgi [
            thru <title> copy p to </title>
        ]
        parse p [thru "Your IP Address is: " copy wan-ip to end]
        write/append servers rejoin [
            b/text " "                          ; username
            wan-ip " "                          ; wan ip
            read join dns:// read dns:// "  "   ; local ip
            j/text "^/"                         ; port
        ]
        alert {
            Server is running.  Connections from clients running on
            your Local Area Network should work without any problems.
            If you want to accept connections from the Internet, and
            you are connected by a router, then the port number you've
            selected must be forwarded from your router to the IP
            address of this computer (see portforward.com for more
            information about forwarding ports).
        }
        focus g
        p: first wait open/lines (join tcp:// j/text)
        z: 1
    ]
    return
    r: area 700x400 rate 4 feel [
        engage: func [f a e][
            if a = 'time and value? 'z [
                if error? try [x: first wait p] [quit]
                r/text: rejoin ["-->  " x "^/" r/text]
                show r
            ]
        ]
    ]
    return
    g: field 400 "Type message here, then press [ENTER]" [
        r/text: rejoin ["<--  " value "^/" r/text]
        show r
        insert p value
        focus face
    ]
    tabs 618
    tab
    button "Save Chat Text" [editor r/text]
    return
]
```

Here's a fun program that lets you record your voice or other sounds to be played as alarms for any number of multiple events. Save and Load event lists. All alarm sounds repeat until stopped. Record

yourself saying 'wake up you lazy bum' or 'hey dude, get up and walk the dog', then set alarms to play those voice messages on any given day/time. If you set the alarm as a date/time, the alarm will go off only once, on that date. If you set the alarm as a time, the alarm will go off every day at that time. The .wav recording code is MS Windows only, but the program can play any wave file that is usable in REBOL:

```
REBOL [title: "Voice Alarms"]

lib: load/library %winmm.dll
mci: make routine! [c [string!] return: [logic!]] lib "mciExecute"

write %play-alarm.r {
    REBOL []
    wait 0
    the-sound: load %tmp.wav
    evnt: load %event.tmp
    if (evnt = []) [evnt: "Test"]
    forever [
        if error? try [
             insert s: open sound:// the-sound wait s close s
        ] [
             alert "Error playing sound!"
        ]
        delay: :00:07
        s: request/timeout [
            join uppercase evnt " alarm - repeats until you click 'stop':"
            "Continue"
            "STOP"
        ] delay
        if s = false [break]
    ]
}

current: rejoin [form now/date newline form now/time]

view center-face layout [
    c: box black 400x200 font-size 50 current rate :00:01 feel [
        engage: func [f a e] [
            if a = 'time [
                c/text: rejoin [form now/date newline form now/time]
                show c
                if error? try [
                    foreach evnt (to-block events/text) [
                        if any [
                             evnt/1 = form rejoin [
                                 now/date {/} now/time
                             ]
                             evnt/1 = form now/time
                        ] [
                            if error? try [
                                 save %event.tmp form evnt/3
                                 write/binary %tmp.wav
                                 read/binary to-file evnt/2
                                 launch %play-alarm.r
                            ] [
                                 alert "Error playing sound!"
                            ]
                            ; request/timeout [(form evnt/3) "Ok"] :00:05
                        ]
                    ]
                ] []  ; do nothing if user is manually editing events
            ]
```

```
        ]
    ]
    h3 "Alarm Events (these CAN be edited manually):"
    events: area  ; {[8:00:00am %alarm1.wav "Test Alarm - DELETE ME"]}
    across
    btn "Record Alarm Sound" [
        mci "open new type waveaudio alias wav"
        mci "record wav"
        request ["*** NOW RECORDING *** Click 'stop' to end:" "STOP"]
        mci "stop wav"
        if error? try [x: first request-file/file/save %alarm1.wav] [
            mci "close wav"
            return
        ]
        mci rejoin ["save wav " to-local-file x]
        mci "close wav"
        request [rejoin ["Here's how " form x " sounds..."] "Listen"]
        if error? try [
            save %event.tmp "test"
            write/binary %tmp.wav
            read/binary to-file x
            launch %play-alarm.r
        ] [
            alert "Error playing sound!"
        ]
    ]
    btn "Add Event" [
        event-name: request-text/title/default "Event Title:" "Event 1"
        the-time: request-text/title/default "Enter a date/time:" rejoin [
            now/date {/} now/time
        ]
        if error? try [set-time: to-date the-time] [
            if error? try [set-time: to-time the-time] [
                alert "Not a valid time!"
                break
            ]
        ]
        my-sound: request-file/title/file ".WAV file:""" %alarm1.wav
        if my-sound = none [break]
        event-block: copy []
        append event-block form the-time
        append event-block my-sound
        append event-block event-name
        either events/text = "" [spacer: ""][spacer: newline]
        events/text: rejoin [events/text spacer (mold event-block)]
        show events
    ]
    btn "Save Events" [
        write to-file request-file/file/save %alarm_events.txt events/text
    ]
    btn "Load Events" [
        if error? try [
            events/text: read to-file request-file/file %alarm_events.txt
        ] [return]
        show events
    ]
]
```

Here are a couple tiny utility scripts that I found useful:

```
; to replace a specific string inside special characters:
```

```
code: "text1 <% replace this %> text3"
replace code "<% replace this %>" "<% text2 %>"
print code

; to replace everything between special characters:

code: "text1 <% replace this %> <% replace this %> text3"
parse/all code [
    any [thru "<%" copy new to "%>" (replace code new " text2 ")] to end
]
print code
```

This code determines the operating system you're running:

```
switch system/version/4 [
    2 [print "OSX"]
    3 [print "Windows"]
    4 [print "Linux"]
    7 [print "FreeBSD"]
    8 [print "NetBSD"]
    9 [print "OpenBSD"]
    10 [print "Solaris"]
] [alert "Can't be dertermined"]
```

Here's a CGI program I keep on my web server to delete masses of email which contain any given "spam" text:

```
#! /home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"Remove Emails"</TITLE></HEAD><BODY>]

spam: [
    {Failure} {Undeliverable} {failed} {Returned Mail} {not be delivered}
    {mail status notification} {Mail Delivery Subsystem} {(Delay)}
]

print "logging in..."
mail: open pop://user:pass@site.com
print "logged in"

while [not tail? mail] [
    either any [
        (find first mail spam/1) (find first mail spam/2)
        (find first mail spam/3) (find first mail spam/4)
        (find first mail spam/5) (find first mail spam/6)
        (find first mail spam/7) (find first mail spam/8)
    ][
        remove mail
        print "removed"
    ][
        mail: next mail
    ]
    print length? mail
]
close mail
print [</BODY></HTML>]
halt
```

The following utility script takes an input string and returns and HTML string with all the web URLs appropriately wrapped as links:

```
bb:  "some text http://guitarz.org http://yahoo.com"
bb_temp: copy bb
append bb_temp " " ; in case the URL doesn't have a trailing space
append bb " "
parse bb [any [thru "http://" copy link to " " (
    replace bb_temp (rejoin [{http://} link]) (rejoin [
    {<a href="} {http://} link {" target=_blank>http://}
    link {</a>}]))] to end
]
bb: copy bb_temp
print bb
```

I use the following utility CGI script to copy entire directories of files from one web server to another:

```
#!/home/path/public_html/rebol/rebol -cs
REBOL []
print "content-type: text/html^/"
print [<HTML><HEAD><TITLE>"wgetter"</TITLE></HEAD><BODY>]
foreach file (read ftp://user:pass@site.com/public_html/path/) [
    print file
    print <BR>
    write/binary (to-file file)
        (read/binary (to-url (rejoin [http://site.com/path/ file])))
]
print [</BODY></HTML>]
```

I've used variations of the following script to rename all the files with a given extension in a folder, to a different extension. The script copies all the files to the same name, without any extension at all:

```
foreach file read %. [
    if (suffix? file) = %.src [
        write (to-file first parse file ".")(read to-file file)
    ]
]
```

I use the following script to keep my collection of Haxe language libraries up to date:

```
REBOL []

write %haxelibs.txt read http://lib.haxe.org/files/

the-list: read/lines %haxelibs.txt
clean: copy []

foreach line the-list [
    x: (parse/all form (find line ".zip") ">")
    if (length? x) > 2 [
        y: parse form second x "<"
        append clean first y
    ]
]

errors: copy []
make-dir %haxelibs
```

```
change-dir %haxelibs
save %list.txt clean  ; comment this if you need to edit list.txt

downloaded: read %.

foreach file clean [
    if not (find downloaded (to-file file)) [
        either error? try [
            size? (join http://lib.haxe.org/files/ file)
        ] [
            print join "ERROR:  " file
            append errors file
        ] [
            print rejoin [
                {Downloading:  } file {  (}
                size? (join http://lib.haxe.org/files/ file) { kb)}
            ]
            if error? try [
                write/binary
                    (to-file file)
                    (read/binary (join http://lib.haxe.org/files/ file))
            ] [
                print join "ERROR:  " file
                append errors file
            ]
        ]
    ]
]

save %haxe_lib_download_errors.txt errors
print "Done."
halt
```

I use the following line to view/edit the code of script which has been run directly from a zip file (compressed folder) in Windows:

```
editor to-file request-file system/script/path
```

Here is a Windows API function to use MCI functions:

```
lib: load/library %winmm.dll
mciSendString: make routine! [
    command [string!]
    rStr [string!]
    cchReturn [integer!]
    hwndCallBack [integer!]
    return: [integer!]
] lib "mciSendStringA"
```

The following scripts demonstrate several different ways to run the code from the action block of another widget (i.e., to simulate mouse clicks or other actions on any given face). To understand more, run "source do-face" and "source do-face-alt" in the REBOL console to see how the "do-face" and "do-alt-face" functions work:

```
view layout [
    button1: btn "Button 1" [alert "Button 1 action block has been run."]
    btn "Button 2" [do-face button1 1]
]
```

```
view layout [
    b1: btn "B1" [alert "B1 left click"] [alert "B1 right click"]
    btn "B2" [do-face b1 1] [do-face-alt b1 1]
]

view layout [
    button1: btn "Button 1" [alert "Button 1 action block has been run."]
    btn "Button 2" [button1/action button1 none]
    ; "button1 none" in the line above releases the down state of the btn
]
```

The following script from http://www.pat665.free.fr/gtk/rebol-view.html demonstrates another way to do the same thing:

```
view layout [
    b: button "Test" [print "Test pressed"]
    button "In" [b/state: true show b]
    button "Out" [b/state: false show b]
    a: button "Action" [
        b/feel/engage :b 'down none
        b/feel/engage :b 'up none
        a/state: false show a ; Not sure why this line is needed...
    ]
]
```

Here is a version of the Windows webcam program from earlier in this tutorial. This version was written for REBOL/face, and includes all the common avicap32.dll constants. It also contains the "do" files required in REBOL/face (they should be deleted if using REBOL/view). It also contains a function that can be used to hide and unhide windows:

```
REBOL []

do %gfx-colors.r
do %gfx-funcs.r
do %view-funcs.r
do %view-vid.r
do %view-edit.r
do %view-feel.r
do %view-images.r
do %view-styles.r
do %view-request.r
do %view.r

;WM_CAP_START: 0x400
;WM_START: to-integer #{00000400}
WM_CAP_START: 1024
WM_CAP_UNICODE_START: WM_CAP_START + 100
WM_CAP_PAL_SAVEA: WM_CAP_START + 81
WM_CAP_PAL_SAVEW: WM_CAP_UNICODE_START + 81
WM_CAP_UNICODE_END: WM_CAP_PAL_SAVEW
WM_CAP_ABORT: WM_CAP_START + 69
WM_CAP_DLG_VIDEOCOMPRESSION: WM_CAP_START + 46
WM_CAP_DLG_VIDEODISPLAY: WM_CAP_START + 43
WM_CAP_DLG_VIDEOFORMAT: WM_CAP_START + 41
WM_CAP_DLG_VIDEOSOURCE: WM_CAP_START + 42
WM_CAP_DRIVER_CONNECT: WM_CAP_START + 10
WM_CAP_DRIVER_DISCONNECT: WM_CAP_START + 11
WM_CAP_DRIVER_GET_CAPS: WM_CAP_START + 14
```

```
WM_CAP_DRIVER_GET_NAMEA: WM_CAP_START + 12
WM_CAP_DRIVER_GET_NAMEW: WM_CAP_UNICODE_START + 12
WM_CAP_DRIVER_GET_VERSIONA: WM_CAP_START + 13
WM_CAP_DRIVER_GET_VERSIONW: WM_CAP_UNICODE_START + 13
WM_CAP_EDIT_COPY: WM_CAP_START + 30
WM_CAP_END: WM_CAP_UNICODE_END
WM_CAP_FILE_ALLOCATE: WM_CAP_START + 22
WM_CAP_FILE_GET_CAPTURE_FILEA: WM_CAP_START + 21
WM_CAP_FILE_GET_CAPTURE_FILEW: WM_CAP_UNICODE_START + 21
WM_CAP_FILE_SAVEASA: WM_CAP_START + 23
WM_CAP_FILE_SAVEASW: WM_CAP_UNICODE_START + 23
WM_CAP_FILE_SAVEDIBA: WM_CAP_START + 25
WM_CAP_FILE_SAVEDIBW: WM_CAP_UNICODE_START + 25
WM_CAP_FILE_SET_CAPTURE_FILEA: WM_CAP_START + 20
WM_CAP_FILE_SET_CAPTURE_FILEW: WM_CAP_UNICODE_START + 20
WM_CAP_FILE_SET_INFOCHUNK: WM_CAP_START + 24
WM_CAP_GET_AUDIOFORMAT: WM_CAP_START + 36
WM_CAP_GET_CAPSTREAMPTR: WM_CAP_START + 1
WM_CAP_GET_MCI_DEVICEA: WM_CAP_START + 67
WM_CAP_GET_MCI_DEVICEW: WM_CAP_UNICODE_START + 67
WM_CAP_GET_SEQUENCE_SETUP: WM_CAP_START + 65
WM_CAP_GET_STATUS: WM_CAP_START + 54
WM_CAP_GET_USER_DATA: WM_CAP_START + 8
WM_CAP_GET_VIDEOFORMAT: WM_CAP_START + 44
WM_CAP_GRAB_FRAME: WM_CAP_START + 60
WM_CAP_GRAB_FRAME_NOSTOP: WM_CAP_START + 61
WM_CAP_PAL_AUTOCREATE: WM_CAP_START + 83
WM_CAP_PAL_MANUALCREATE: WM_CAP_START + 84
WM_CAP_PAL_OPENA: WM_CAP_START + 80
WM_CAP_PAL_OPENW: WM_CAP_UNICODE_START + 80
WM_CAP_PAL_PASTE: WM_CAP_START + 82
WM_CAP_SEQUENCE: WM_CAP_START + 62
WM_CAP_SEQUENCE_NOFILE: WM_CAP_START + 63
WM_CAP_SET_AUDIOFORMAT: WM_CAP_START + 35
WM_CAP_SET_CALLBACK_CAPCONTROL: WM_CAP_START + 85
WM_CAP_SET_CALLBACK_ERRORA: WM_CAP_START + 2
WM_CAP_SET_CALLBACK_ERRORW: WM_CAP_UNICODE_START + 2
WM_CAP_SET_CALLBACK_FRAME: WM_CAP_START + 5
WM_CAP_SET_CALLBACK_STATUSA: WM_CAP_START + 3
WM_CAP_SET_CALLBACK_STATUSW: WM_CAP_UNICODE_START + 3
WM_CAP_SET_CALLBACK_VIDEOSTREAM: WM_CAP_START + 6
WM_CAP_SET_CALLBACK_WAVESTREAM: WM_CAP_START + 7
WM_CAP_SET_CALLBACK_YIELD: WM_CAP_START + 4
WM_CAP_SET_MCI_DEVICEA: WM_CAP_START + 66
WM_CAP_SET_MCI_DEVICEW: WM_CAP_UNICODE_START + 66
WM_CAP_SET_OVERLAY: WM_CAP_START + 51
WM_CAP_SET_PREVIEW: WM_CAP_START + 50
WM_CAP_SET_PREVIEWRATE: WM_CAP_START + 52
WM_CAP_SET_SCALE: WM_CAP_START + 53
WM_CAP_SET_SCROLL: WM_CAP_START + 55
WM_CAP_SET_SEQUENCE_SETUP: WM_CAP_START + 64
WM_CAP_SET_USER_DATA: WM_CAP_START + 9
WM_CAP_SET_VIDEOFORMAT: WM_CAP_START + 45
WM_CAP_SINGLE_FRAME: WM_CAP_START + 72
WM_CAP_SINGLE_FRAME_CLOSE: WM_CAP_START + 71
WM_CAP_SINGLE_FRAME_OPEN: WM_CAP_START + 70
WM_CAP_STOP: WM_CAP_START + 68

avicap32.dll: load/library %avicap32.dll
user32.dll: load/library %user32.dll

; Hide rebface console:
```

```
get-focus: make routine! [return: [int]] user32.dll "GetFocus"
hwnd-hide-console: get-focus
hide-window: make routine! [
    hwnd [int]
    a [int]
    return: [int]
] user32.dll "ShowWindow"
hide-window hwnd-hide-console 0

view/new center-face layout/tight [
    image 320x240
    across
    btn "Take Snapshot" [
        sendmessage cap-result WM_CAP_GRAB_FRAME_NOSTOP 0 0
        sendmessage-file cap-result WM_CAP_FILE_SAVEDIBA 0 "scrshot.bmp"
    ]
    btn "Exit" [
        sendmessage cap-result WM_CAP_END 0 0
        sendmessage cap-result WM_CAP_DRIVER_DISCONNECT 0 0
        free user32.dll
        quit
    ]
]

; Set window title:

set-caption: make routine! [
    hwnd [int]
    a [string!]
    return: [int]
] user32.dll "SetWindowTextA"
hwnd-set-title: get-focus
set-caption hwnd-set-title "Web Camera"

find-window-by-class: make routine! [
    ClassName   [string!]
    WindowName  [integer!]
    return:     [integer!]
] user32.dll "FindWindowA"
hwnd: find-window-by-class "REBOLWind" 0

cap: make routine! [
    cap [string!]
    child-val1 [integer!]
    val2 [integer!]
    val3 [integer!]
    width [integer!]
    height [integer!]
    handle [integer!]
    val4 [integer!]
    return: [integer!]
] avicap32.dll "capCreateCaptureWindowA"
sendmessage: make routine! [
    hWnd [integer!]
    val1 [integer!]
    val2 [integer!]
    val3 [integer!]
    return: [integer!]
] user32.dll "SendMessageA"

sendmessage-file: make routine! [
    hWnd [integer!]
    val1 [integer!]
```

```
    val2 [integer!]
    val3 [string!]
    return: [integer!]
] user32.dll  "SendMessageA"

cap-result: cap "cap" 1342177280 0 0 320 240 hwnd 0
; 1342177280 in the line above is the value I got from
; BitOR(WS_CHILD,WS_VISIBLE) in two seperate development environments,
; but not sure if it will always hold true.
sendmessage cap-result WM_CAP_DRIVER_CONNECT 0 0
sendmessage cap-result WM_CAP_SET_SCALE 1 0
sendmessage cap-result WM_CAP_SET_OVERLAY 1 0
sendmessage cap-result WM_CAP_SET_PREVIEW 1 0
sendmessage cap-result WM_CAP_SET_PREVIEWRATE 1 0

do-events
```

Here's one final version of the web cam program, with a nicer save feature. In order for the save routine to work properly, this code should be saved to a .r script and run from there:

```
REBOL []

avicap32.dll: load/library %avicap32.dll
user32.dll: load/library %user32.dll
get-focus: make routine! [return: [int]] user32.dll "GetFocus"
set-caption: make routine! [
    hwnd [int] a [string!]  return: [int]
] user32.dll "SetWindowTextA"
find-window-by-class: make routine! [
    ClassName [string!] WindowName [integer!] return: [integer!]
] user32.dll "FindWindowA"
sendmessage: make routine! [
    hWnd [integer!] val1 [integer!] val2 [integer!] val3 [integer!]
    return: [integer!]
] user32.dll "SendMessageA"
sendmessage-file: make routine! [
    hWnd [integer!] val1 [integer!] val2 [integer!] val3 [string!]
    return: [integer!]
] user32.dll  "SendMessageA"
cap: make routine! [
    cap [string!] child-val1 [integer!] val2 [integer!] val3 [integer!]
    width [integer!] height [integer!] handle [integer!]
    val4 [integer!] return: [integer!]
] avicap32.dll "capCreateCaptureWindowA"

view/new center-face layout/tight [
    image 320x240
    across
    btn "Take Snapshot" [
        sendmessage cap-result 1085 0 0
        sendmessage-file cap-result 1049 0 "scrshot.bmp"
        save-path: first split-path system/options/script
        view/new center-face layout [
            image load join save-path %scrshot.bmp
            btn "save" [
                (write/binary
                    to-file pp: request-file/save/file %photo1.bmp
                    read/binary join save-path %scrshot.bmp
                )
                alert join "Saved " pp
                unview
```

```
            ]
        ]
    ]
    btn "Exit" [
        sendmessage cap-result 1205 0 0
        sendmessage cap-result 1035 0 0
        free user32.dll
        quit
    ]
]
hwnd-set-title: get-focus
set-caption hwnd-set-title "Web Camera"  ; title bar
hwnd: find-window-by-class "REBOLWind" 0
cap-result: cap "cap" 1342177280 0 0 320 240 hwnd 0
sendmessage cap-result 1034 0 0
sendmessage cap-result 1077 1 0
sendmessage cap-result 1075 1 0
sendmessage cap-result 1074 1 0
sendmessage cap-result 1076 1 0
do-events
```

The following code demonstrates how to check for async keystrokes (including arrow keys) in the REBOL shell:

```
print ""
p: open/binary/no-wait console://
q: open/binary/no-wait [scheme: 'console]

forever [
    if not none? wait/all [q :00:00.30] [
        wait q
        qq: to string! copy q
        probe qq
    ]
]
```

Don't forget to look at all the scripts at rebol.org - not just in the scripts section, but also in the email and AltME archives. It's a treasure trove of working code examples and answers to virtually any coding problem!

12. Learning More About REBOL - IMPORTANT DOCUMENTATION LINKS

A very old edition of this text with several hundred screen shot images is available at http://musiclessonz.com/rebol_tutorial-images.html). If you're completely new to programming, that text may offer some helpful simple perspective.

The tutorial at http://www.rebol.com/docs/rebol-tutorial-3109.pdf provides a nice summary of fundamental concepts. It's a great document to read next. To learn REBOL in earnest, read the REBOL core users manual: http://rebol.com/docs/core23/rebolcore.html. It covers all of the data types, built-in word functions and ways of dealing with data that make up the REBOL/Core language (but not the graphic extensions in View). It also includes many basic examples of code that you can use in your programs to complete common programmatic tasks. Also, be sure to keep the REBOL function dictionary handy whenever you write any REBOL code: http://rebol.com /docs/dictionary.html. It defines all the words in the REBOL language and their specific syntax use. The dictionary is also helpful in cross-referencing function words that do related actions in the language (great when you can't remember a function name you're looking for). Along the way, read the REBOL View and VID documents at: http://rebol.com/docs/easy-vid.html , http://rebol.com /docs/view-guide.html , http://rebol.com/docs/view-system.html , http://www.rebol.com/how-to /feel.html , http://www.pat665.free.fr/gtk/rebol-view.html , and run the script at http://www.rebol.org /download-a-script.r?script-name=vid-usage.r. Those documents explain how to write Graphical User Interfaces in REBOL. Once you've got an understanding of the grammar and vocabulary of the language, dive into the REBOL cookbook: http://www.rebol.net/cookbook/. It contains many simple and useful examples of code needed to create real-world applications. When you've read all that, finish the rest of the documents at http://rebol.com/docs.html.

Beyond the basic documentation, there is a library of hundreds of commented REBOL scripts at http://rebol.org. There's also a searchable archive of the mailing list and AltME (community forum) containing several hundred thousand posts at rebol.org. That archive contains answers to many thousands of questions encountered by REBOL programmers. Rebol.org is an essential resource! There are numerous other web sites such as http://www.codeconscious.com/rebol, http://www.rebolforces.com (duplicated at http://www.rebolplanet.com), http://www.reboltech.com /library/library.html, http://www.fm.vslib.cz/~ladislav/rebol, http://www.compkarori.com/vanilla/display /index, http://www.rebol.net, http://reboltutorial.com, http://blog.revolucent.net/search/label/REBOL, http://www.reboltalk.com/forum, http://anton.wildit.net.au/rebol, http://rebolweek.blogspot.com, http://groups-beta.google.com/group/Rebol, and rebolfrance (translated by Google) that provide more help in understanding and using the language. Don't miss Carl Sassenrath's personal blog, *discussions about REBOL3*, alpha downloads of REBOL3, and REBOL3 documentation. For a complete list of all web pages and articles related to REBOL, see http://dmoz.org/Computers /Programming/Languages/REBOL/.

Don't forget to click the rebsite icons in the "REBOL" and "Public" folders, right in the desktop of the REBOL interpreter. Right-click any of the hundreds of individual program icons and select "edit" to see the code for any example. That's a great way to see how to do things in REBOL.

13. Beyond REBOL

Modern computers are complex systems built upon multiple layers of technology. The physical hardware (CPU, RAM memory, hard drive, keyboard, mouse, monitor, etc.) form the foundation. The operating system (Windows, Mac, Linux, etc.) manages that hardware, enables software drivers, provides a common user interface, and provides many basic facilities to make the whole system useful (file management, connection to network protocols, etc.). Software built upon the fundamental components in the operating system make more specific applications possible (word processors, games, etc). In our modern world, many of the applications we use are built upon *multiple* software layers, on top of the already complex foundation. The Internet is made up of many types of hardware systems, running many different operating systems, connected by compatible network protocols, running many different web and email server programs, storing information via database programs of all types, etc. All those layers work together to serve data via generally compatible formats (HTML files containing page layouts, standard image types such as .jpg and .gif, standard sound formats such as .mp3 and .wav, and standard video formats such as Flash). That's all accessed by a variety of different web browser programs, email clients, cell phone apps, etc., which connect to those standard protocols through the OS, and read/save info in those formats. On top of that complex structure, languages like Javascript run within web browser software to control data which appears on web pages. Languages like PHP and others run on web server software to control how they output data.

REBOL is a language that operates at many of those levels. It can run as a browser plug-in to control data display in web pages. It can run on a web server to build and serve web sites. It neatly "wraps" up most common functions that various operating systems enable, to provide file handling, network control, and other system level facilities. It provides a single, simple format that lets you talk to all different computers in the same ways, at all those levels. It's got it's own way of speaking that is different from many other languages. That grammar and vocabulary is called the "API". If you continue to pursue programming in various environments, you'll encounter many different language APIs which, in the end, do most of the same things as REBOL, but which use very different approaches to grammar and syntax. Eventually, you'll learn to deal with the raw API of the operating system (using native language compilers, DLLs, and other native interfaces). The operating system API is the base language that most other languages are actually *translating* to. Because the operating system needs to access the computer hardware quickly, it is written in a "lower level" language - one that is formatted to think more like the computer's raw calculations, and less like human speech.

With REBOL, you can do most typical things that programmers want to do, but there are many functions in the various operating system APIs that aren't included (i.e., web cam access, sound input, low level hardware control, etc.). To do that, be prepared to explore the raw operating system API, and the language(s) in which it was written. On Windows, Unix, Macintosh, and other platforms, that typically means learning the syntax and structure of the "C" and "C++" languages. Also, learning common methods for accessing shared code files such as .dll's is very important. Once you've learned the full REBOL API, that's a good direction to take in your studies.

Other favorite programming languages of this author, which pack a lot of computing punch, like Rebol, include:

1. Haxe/Neko - compiles your code to several different languages/platforms. It runs on Windows, Mac, and Linux, using the exact same code. It contains the entire Flash Actionscript3 API, and can compile directly to standard .swf files, which makes it extraordinarily powerful for creating multimedia applications, for use in both online and desktop applications. Haxe can compile to the Neko virtual machine, for use in server and desktop applications. Neko applications can be converted directly to native Windows, Mac and Linux executables. Haxe can also compile directly to Javascript, PHP, *and* C++ code, all using the exact same core language. It uses a traditional syntax familiar to those who know Java and C++. It's a very small download, runs extremely fast, is free/open source, and is very stable. Haxe is a great tool to compliment REBOL, because it's strengths cover some of REBOL's weaknesses (Flash multimedia development and integration with other popular high level *and* low level development tools). Mtasc is another free Flash compiler, created by the same person as Haxe. It's older, but may be useful if you want to compile code written in the Actionscript2 API. Openlaszlo is one more free, cross-platform tool for those interested in developing rich multimedia web applications. It

has its own language implementation (different from the Flash Actionscript API), but can compile the exact same code to either Flash or DHTML, so applications written in Openlaszlo can run in virtually any web environment.

2. JAVA - The most popular programming language around. If you want a job programming, JAVA should be in your short list of languages to learn. Programs written in JAVA can run on Windows, Mac, Linux, cell phones, web browsers, and most other modern operating platforms, using the same code. JAVA has tens of millions of users, so support for it is enormous, and integration with other tools is ubiquitous. The overwhelming majority of desktop computers already have the JAVA virtual machine installed, and you can accomplish just about any programming goal with JAVA tools. One down side of JAVA is that it's much larger and more complex (both in language structure and download size) than REBOL and other tools. You'll need to learn more about traditional object oriented design patterns to work with JAVA.
3. Python - Another very popular free/open source programming tool that runs on most operating systems. It's smaller and easier to learn than JAVA, but is still powerful and has strong support around the world. It's great for creating web site scripts as well as desktop apps of all sorts. Python covers much of the same problem domain, and has some size/simplicity features similar to REBOL (although REBOL is much smaller and simpler to use :).
4. Purebasic - A nice compiler that creates very small and fast native applications for Windows, Mac, and Linux. It's not free, but it is inexpensive, and upgrades are free for life. Purebasic offers many of the benefits of programming with lower level languages such as assembler and C/C++ (execution speed, and access to low level optimization). It comes with a very nice integrated development environment and makes use of a friendly and very productive cross platform language implementation.
5. AutoIt - The unique characteristic of Autoit is that it includes many built-in functions to **control *other* Windows programs**. You can programatically push buttons, type text, select menu items, choose items from lists, etc. in any program window, as if those actions had been performed by a user clicking and typing on screen. This allows you to automate and speed up repetitive routines, and to customize the use of existing applications. AutoIt is extremely simple to learn, it's free and quick to download/install, has a large user base, easily compiles scripts to standard .exe programs, and is a powerful general purpose scripting language that can be used to create all types of applications for Windows.
6. If your goal is to work as a commercial programmer, you should become fluent with the most popular tools. To work with development teams, you need to know the language(s) they use. The list at http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html will point you in the right direction (C/C++, C#, Visual Basic, PHP, PERL, etc.). You'll also need to know HTML and Javascript if you want to do any work as a web developer.

Exploring those tools should give focus to your further studies. Good luck and have fun!

For feedback, bugs reports, suggestions, etc., please email Nick at:

```
reverse {moc tod znosselcisum ta lober}
```

To hire the author for tutoring, to develop software or web site applications, or for general consultation/computing support, see http://com-pute.com or call 267-352-3625.

14. Appendix 1: A REBOL Song

You can find an MP3 of the following REBOL song at http://re-bol.com/examples/rebol_song.mp3:

```
I've got some operators and a little block code
And with VID I'm gonna build a new window.
I'm gonna hack it out, simple and fast
And make a killer REBOL app that lasts.

They tell me I should give Python a try
But they don't know how that'd make me cry.
It takes ten times as long to install that mess
And still gives an error at some memory address.

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

Gonna take some strings and put 'em in a series
And loop through so fast, it'll bring you to your knees.
'Bind, 'set, 'use, 'context, and 'get
Once you get how they work, they're no sweat.

I never felt this way about anything else
Not C++, JAVA, or Haskell.
It's 2am, and I'm almost done
but I'm codin' on, cause it's so much fun.

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

Regular expressions in PERL are alright
But let me loose with parse - I can do that all night.
PHP's ok, for data on the net
but nothing's slicker than an IOS Reblet.

"Foreach" 100 lines they have to code
I just use a dialect to lighten my load.
And just when I'm stuck on the toughest part
Some guru writes code that looks like REBOL art.

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

Delphi, C, Ruby, LUA - no more.
When REBOL came along, they went out the door.
You can't get a thing done with those old dogs
And I don't want to run another resource hog.

PC, MAC, Linux - they all work for me
Heck, I can even do OpenBSD.
We even have a plug-in for your browser
Ooh look, those BASIC guys just peed their trowsers :)

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
```

```
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

When you need function help, who wants to grab a book,
It's built right into REBOL - you know where to look.
One half a meg - that's all it takes
To get rid of all your coding headaches.

It may be tough to convince your boss
To try out REBOL, even though it's not FOSS.
But the best things in life are still free.
It's just the REBOL source, that we'll never see

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

I don't need a single library
for all the protocols and datatypes I ever need.
Build a GUI in a line or 2
Who needs modules, I just type "do".

Productivity they can't comprehend,
with REBOL I do the work of 10 men.
I don't wanna code any other way
Just write .r scripts every day.

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath

Now I spend all my time on AltME
While I pour another cup of black coffee
Another job done, I'm crankin' 'em out
Just wish that REBOL could make a darn printout.

I never said it's perfect, but I don't care
REBOL keeps me from pulling out my hair.
Without it I'd really have to give up
And install some 100 megabyte setup.

REBOL REBOL - Get it done fast
R-E-B-O-L   - and have a blast
REBOL REBOL - It's in my path
Thanks Mr. Sassenrath
```