Learn RFO Basic - The Easiest Way To Create Android Apps

By: Nick Antonaccio
Updated: 2-26-2013 (RFO Basic! version 1.71 or higher is required for this tutorial)
Learn the simplest and most productive development tool for Android.
There's absolutely no easier way to create full featured apps for Android phones and devices.
Visit http://rfobasic.freeforums.org to ask questions about RFO Basic.
Go to http://laughton.com/basic/versions/index.html to download RFO Basic.
See De Re Basic! for API reference documentation.
Check out easiestprogramminglanguage.com and learnrebol.com for a related in-depth desktop programming tutorials about the REBOL language, and freeconsignmentsoftware.com for a project written in it.

Contents:

1. Introducing RFO Basic!

What is RFO Basic? Why use it?

- RFO Basic is a uniquely *simple* and *productive* development tool that can be used to create powerful apps for Android phones and devices. There's absolutely no easier way to program applications for Android.
- RFO Basic runs entirely on your Android device. It's a tiny, self contained, on-device programming solution, which *doesn't require any software installed on a desktop computer*. You *can* create RFO Basic applications on your desktop PC, and if you install the Eclipse IDE and Android API, you'll be able to produce full-fledged Android APK files ("apps", "programs") which can be distributed in the Android app store. For those who want the simplest possible deployment solution, there's a tiny free program for Windows which automatically creates APKs for you. No additional knowledge of the Android environment is required.
- RFO Basic is *powerful*. It provides access to **hardware, sensors, sound, graphics, multitouch, file system, SQLite, network sockets, FTP, HTTP, bluetooth, HTML GUI, encryption, SMS, phone, email, text-to-speech, voice recognition, GPS, math, string functions, list functions**, etc. Unlike other flavors of BASIC which are limited to introducing fundamental programming concepts, RFO Basic is a rich, modern, featured-filled language. Although simple to use, it's no toy, but a practical tool for creating useful Android software.
- RFO Basic is 100% *free* and open source.
- RFO Basic is supported by a friendly and knowledgeable community of active developers around the world. It's regularly improved, with updates typically occuring several times a month.
- RFO Basic has useful *built-in help* for all available functions and language constructs, and several dozen complete example programs, immediately available directly on your Android device.
- RFO Basic is *ridiculously simple to learn and use*. There are no deep programming language concepts to ingrain, there's no complex background knowledge to understand, no enormous desktop IDE to install, no unwieldy Android SDK to learn, no JAVA language cruft to muddy code, no app store approvals to request, or any of the typical hurdles encountered when creating programs for mobile devices. Just download the 1/2 Meg RFO Basic app to your phone, familiarize yourself with the included function reference and example programs, and start writing simple command based procedural code in a text editor. There are dozens of concise functions to manage data, to control hardware, to interact with users, and to quickly complete many types of *useful* work. The interpreter/runtime installs in less than a minute, and you can set up a desktop PC as a code editing machine with an additional couple minutes.
- RFO Basic is *easy* enough for absolute beginners and average computer users to operate immediately, but *powerful* enough for a variety of professional uses. You can treat it as a simple personal calculator or utility app for your phone, or as a full blown development environment to create marketable apps. You can write quick scripts on your device to manage files, view and edit database info, process text, download web data, perform FTP uploads, take photos, process images, easily access voice recognition and other Android features, etc., or you can use it to develop graphic games or network applications which you can sell in the market - and everything in between. (Be aware that RFO Basic is GPL licensed, so if you release any commercial software which includes its code, you must also release the source).
- Experienced developers and IT professionals will find that RFO Basic allows a wide variety of utility scripts to be developed and deployed in minutes and hours, rather than days and weeks, and with a dramatically short learning curve.
- RFO Basic can be picked up immediately by anyone who's used any form of traditional Basic. An entire community of developers and novice programmers can leverage their existing skills in this

familiar and easy environment, to build *useful* mobile apps.

If you're new to Basic or even to programming in general, don't fret. RFO Basic is an *extremely simple* subset of other familiar Basic programming languages. You can read the entire API and the reference documentation in a single day. This tutorial will provide enough fundamental understanding, working code examples, and case studies to get you creating apps of all types, completely on your own.

2. Getting Started: Downloading and Installing RFO Basic, Hello World

The RFO Basic interpreter is an app (software program) that runs on your Android device. It translates written text in the RFO Basic language syntax ("source code") into apps which other users can run on their device. To get the free RFO Basic application, go to:

http://www.laughton.com/basic/versions/index.html

Go to the link above in your *phone's web browser*, download the most recent version of the *Basic.apk* file to your Android device, and run the installation (alternately, you could choose to install using the Google Play store). Once you've got RFO Basic installed, click the "Basic!" app icon to run it, press the Android Menu button on your phone, select "Clear", and type the following code into the text area:

```
print "Hello world!"
```

To execute the code, press the Android Menu button again, and select "Run".

Before going any further, give it a try. Download RFO Basic and type in the code above to see how it works. It's extraordinarily simple and literally takes just a few seconds to set up. To benefit from this tutorial, it's critical that you *type or paste each code example into the RFO Basic interpreter* to see what happens.

3. Several Simple Examples

3.1 Tiny Note Pad App

To get used to working with RFO Basic, and to see how simple the code is, try this:

Run the RFO Basic app, press the Android Menu button, select "Clear", and type the following code into the text editor area:

```
file.exists b, "temp.txt"
if b=0 then console.save "temp.txt"
grabfile s$, "temp.txt"
text.input r$, s$
print r$
console.save "temp.txt"
```

This program first checks to see if the text file "temp.txt" exists on your Android device. If the "temp.txt" file does not exist, it creates a new blank text file named "temp.txt". Then it reads the contents of the "temp.txt" file, and allows the user to edit it. Finally, it saves the edited text back to the file "temp.txt". This tiny program is a real, useful notepad app that you can use to jot down and save important text information of any kind.

To save the program code above, press the Android Menu button, select "Save", and enter a file name (i.e., "edit.bas"). Run the program by pressing the Android Menu button and select "Run". You can load, edit, and run this saved program at any time in RFO Basic by pressing the Android Menu button, and selecting "Load".

If you have any trouble, you can download the code above from http://www.rfobasic.com/edit.bas. You can try a working version of this app at http://www.rfobasic.com/edit.apk.

3.2 Teeny Timer App

Now, create a new blank .bas file (Basic! -> Menu Button -> Clear) and type the following code:

```
s = clock()
print "Timer started at: "; s
do
    inkey$ k$
    if ((k$ <> "@") & (k$ <> "key 4")) then let done = 1
until done
e = clock()
print "TIME: "; (e - s) / 1000; " seconds"
```

Save and run the program. It's a useful stopwatch app, precise to thousandths of a second. If you have any trouble with the example, you can download and run the completed code from http://www.rfobasic.com/timer.bas. You can try a working version of this app at http://www.rfobasic.com /timer.apk.

As you can see, useful RFO Basic programs can be *very* short and simple to create.

3.3 Online Examples

There are a number of interesting example programs created by community members at http://laughton.com/basic/programs/. You can get to know a lot about RFO basic just by exploring the downloadable code in those apps. By default, any downloaded RFO Basic program code should be saved in the "/sdcard/rfo-basic/source/" folder on your Android device. Any downloaded resources required to run the programs (images, configuration files, etc.) should be put in the "/sdcard/rfo-basic/data/" folder. SQLite databases should be put in the "/sdcard/rfo-basic/databases" folder. More information about code donated by the RFO Basic community is available at http://rfobasic.freeforums.org/shared-program-instructions-7-30-11-t125.html.

3.4 Editing and Workflow Tips

On many small Android devices, the text editor built into RFO Basic is not the most comfortable or productive environment for editing code. It's also often helpful to copy/paste code from files or web sites (such as this tutorial) from your desktop computer, directly into the RFO Basic interpreter. One way to do that is to use an Android Emulator that runs on your desktop computer. Youwave and Bluestacks are two emulators that work with RFO Basic in Windows and Mac OS. Unfortunately, emulators are large programs that use many system resources, so they may run slowly on all but the most powerful desktop computers.

A more efficient option is to connect your Android device to your desktop computer using a USB cable, and set the connection type to Disk Drive. With this done, you can edit and save program code using any text editor, then reload and run it in RFO Basic.

A wireless option is to use programs such as Wifi File Transfer to set up a server on your phone, then connect using the web browser on your desktop PC, to transfer edited files. This is an easy way to get code back and forth between your Android and PC wirelessly, without setting up any software on the desktop machine.

Android file explorer apps such as the free ES File Explorer provide yet another way to edit and tranfer code. Upload your edited code files to an FTP server or save to a shared network folder (on your desktop PC, right click any folder and select "share with", to share it on your local area network). You can then open the files directly with ES File Explorer - it provides simple direct loading and editing of files on FTP servers and LAN shares. From the ES file editor, you can copy/paste code into the RFO Basic editor. This can be a useful option if several people need to share/edit the same code.

If you do a lot of code editing directly *on* your Android device, it can be helpful to use a software keyboard such as A.I. Type, which supports Undo/Redo, cursor keys, and other useful editing features. This keyboard can be used directly in the RFO Basic code editor, but other editors such as Jota and DroidEdit provide similar features.

The tool preferred by this author is Wifi File Transfer. Edit code using your favorite desktop text editor, save and transfer to the /sdcard/rfo-basic/source/ folder on your Android using Wifi File Transfer, then (re)open and run it with RFO Basic. (Be sure to check "Overwrite existing files" when using this process). This procedure is simple, quick, and allows you to copy/paste/edit/run back and forth between any desktop PC and Android device in only a few seconds, without wires, and without *any* software configuration on the desktop (Wifi File Transfer does *not* require shared folders, login credententials, etc. - just a browser and a network connection).

In order to execute the numerous Edit/Transfer/Run cycles required to write and debug any type of code, it's essential to become perfectly at home (and FAST) with at least one of these routines.

4. RFO Basic Language Fundamentals

4.1 Functions/Commands

As with any modern programming language, to use RFO Basic, you need to learn how to use **"commands"** (alternately called "functions" or "methods", in various programming languages). Commands are words that perform actions. They typically process data in some way. Command words are followed by data **"parameters"** (also called "arguments"). In RFO Basic, multiple parameters are separated by commas. Type and run this command in the RFO Basic code window, to see how it works:

```
popup "Hello world!",0,0,1
```

Notice how changing the parameters alters how the command performs:

```
popup "Moved over, and shorter duration...",50,50,0
```

The "print" command is very important. You can print a single item, either a number or text:

```
print "five"
print 5
```

You can print multiple items, separated by commas:

```
print "one", "two", "three"
```

You can print multiple items right next to each other, using semicolons:

```
print "one"; "two"; "three"
```

If you end a print command with a semicolon, RFO Basic will wait for the next print command, to actually display any data:

```
print "waiting...";
pause 2000
print "done"
```

Another immediately useful command is "console.save". Console.save saves the console output of RFO Basic (anything that has been printed in the current program), to the specified file:

```
print "test"
console.save "temp.txt"
```

Many of the commands in RFO Basic follow the format: category.action. For example, GPS command "gps.latitude" gets your current latitude, "gps.altitude" gets your current altitude. The audio command "audio.play" begins playing a sound, and "audio.record" begins recording a sound, etc.

*Note: technically, there is a difference between a "command" and a "function" in RFO Basic (functions require parentheses and always return a value), but for the purposes of this tutorial, we'll use the term **"function"** to refer to all action words, functions, and commands.*

4.2 Comments

Comments are small human readable notes, added to programs, which are completely ignored by the computer language interpreter. Their purpose is to remind programmers what a particular piece of code does, and they are meant to be only human-readable. In RFO Basic, there are three types of comments:

```
print "Hello"   % all text after a percent sign is IGNORED

! Exclamation points force entire lines to be ignored

!!
You can comment any number of multiple lines by surrounding
them with double exclamation points.
Nothing in this program is executed, except for the first
'print "Hello"' code.
!!
```

4.3 Variables

Variables are words that a programmer comes up with to *label* and represent stored data, so that the data can be referred to and used again later in a program. To create a variable label, programmers choose any combination of letters and numbers, as long as the variable starts with a letter, and contains no special characters. Text, or *"string"* variables in RFO Basic are typically followed by the "$" character. A word followed by an "=" sign sets a variable label:

```
person$ = "John"
```

Along with functions, variables are perhaps the most fundamental and pervasive concepts in all programming, regardless of language. After the code above has been executed, the label "person$" can be used anywhere (without the equal sign), to represent the text "John":

```
popup person$,0,0,1
```

Variable values can be *changed* at any point in a program - that is one of the main reasons for their use:

```
person$ = "Dave"
popup person$,0,0,1
```

In RFO Basic, variables are NOT case sensitive:

```
popup person$,0,0,1
popup PERSON$,0,0,1
popup PeRsOn$,0,0,1

! to the RFO Basic interpreter, the three lines above are the same
```

Variables are extremely important in every type of programming because they represent changable ("variable") data. Data input by a user, read from a file, returned by a sensor, etc., can all be represented by variables. You could store the entire text of a book read from a web server into a variable labeled "book$". You could read the file names contained in a folder on your SD card and give that list the label "files$". You could ask your user to select an MP3 from your file system and assign it the variable label "song$". In any of these cases, you can use the varaiable labels you've given the data to load, save, display, manipulate and otherwise process the represented data, all using a single variable word.

4.4 Function Return Values

Most functions produce **"return"** values. Return values are the *output* data produced when a function processes any given input data. Data processed and returned by a function is typically stored in variables:

```
lowered = lower$ ("THIS FUNCTION CONVERTS TEXT TO LOWERCASE.")
print lowered
```

In some cases the output (return value) of one function can be used directly as the input of another function:

```
print lower$ ("THIS FUNCTION CONVERTS TEXT TO LOWERCASE.")

! The previous line contains 2 functions: 'print' and 'lower$'
! The output of the lower$ function is used as the input
! parameter of the print function.
```

In RFO Basic, **the return value(s) of functions (commands) are most often specified as *parameter(s)* - placed in the parameter list, immediately *after* the function word**. The following function "grabfile" stores the text read from the specified file, in the arbitrarily chosen variable "s$":

```
print "Hello World!"        % this prints some text
console.save "temp.txt"     % this saves the printed text to a file
grabfile s$, "temp.txt"     % this reads the file and stores the
                            %   returned data in the variable s$
print s$                    % this prints the contents of the variable
```

The return values output by the numerous functions in RFO Basic can be used in your programs to accomplish useful goals. The "input" function allows you to request a single line of text, or a number, from the user:

```
input "Enter some text", s$   %text input by user is stored in variable s$
print s$
```

Notice the following line contains the return value "n", *WITHOUT THE "$" CHARACTER. N is therefore treated as a **number** variable*:

```
input "Enter a number", n
print n
```

The "input" function has an *optional* parameter that allows you to display some default text in the requester:

```
input "Enter your name", s$, "John"   % the "$" in s$ means that it's text
print s$
```

The "text.input" function allows the user to input, edit, and save large quantities of text. The first parameter is a string variable that holds the final edited text output by the function. The second parameter holds some initial text to appear in the editor window:

```
text.input r$, "Edit this text"       % user edited text is stored in r$
print r$
```

You can assign a variable to text read from a file (using the grabfile function above), and use that text as the second parameter in the text.input function. This allows you to retrieve, edit, and store any text file on your Android device:

```
grabfile s$, "temp.txt"   % The variable s$ is assigned to the text read
                          %   from the file "temp.txt"
text.input r$, s$         % The initial text in the editor is the text
                          %   stored above in the s$ variable.  After the
                          %   user edits the text, the returned, edited
                          %   text is stored in the r$ variable
```

4.5 A Variety of Useful Function Examples

Here are a few more examples which demonstrate how to perform useful actions and how to process data in useful ways, using functions/commands. Try typing or pasting every one of them into the RFO Basic interpreter, to see how they work:

```
! The tone function takes 2 parameters, pitch and duration, and plays a
! tone:

    tone 440, 2000

! The file.rename function takes 2 parameters, the old and new filenames,
! and renames the old file to the new file name in the Android file
! system:

    file.rename "oldname.txt", "newname.txt"

! The clipboard.get function takes 1 parameter, a variable name assigned
! to the contents retrieved from the device clipboard:

    clipboard.get c$
    print c$         % this prints the data returned by the function above

! The tts.init function prepares for the use of text-to-speach functions
! No parameter is required:
    tts.init

! The tts.speak function takes 1 parameter, some text to speak:

    tts.speak "I am alive"

! This code reads the current text in the Android clipboard:

    clipboard.get c$
    tts.init
    tts.speak c$

! The audio.record.start takes 1 parameter, the name of a file to record
! to:

    audio.record.start "record.3gp"

! The pause function takes 1 parameter, the amount of time to wait (in
! milliseconds):

    pause 5000

! The audio.record.stop function doesn't take any parameters.  It just
! stops the recording previously started:

    audio.record.stop

! The audio.load function takes 2 parameters, the name of a "pointer"
! number used to refer to a specified audio file, and the name of the
! specified audio file:

    audio.load s, "record.3gp"

! The audio.play function plays the file specified by the pointer number
! created in the previous function:

    audio.play s
    pause 5000

! The clock() function doesn't take any parameter.  It returns the number
! of milliseconds since your device was turned on:

    s = clock()

! This prints the data contained in the numeric variable created above:

    print s

! The device function takes 1 parameter, the name of a string variable to
! store returned information about your device:

    device s$

! This line prints the data contained in the string variable created
! above:

    print s$

! The gps.open function doesn't take any parameters.  It just prepares for
! the use of the GPS function:

    gps.open
```

```
! The gps.latitude and gps.longitude functions each take 1 parameter,
! a variable name to store the returned latitude and longitude values:

    gps.latitude l1
    gps.longitude l2

! The gps.close function doesn't take any parameters.  It just closes
! the gps device opened above:

    gps.close

! This prints the longitude and latitude values stored in the return
! variables created above:

    print "Your longitude and latitude:  "; l1; ", "; l2

! This assigns some text to a new string variable:

    s$ = "three blind mice"

! The replace function takes 3 parameters, the name of an input AND return
! string variable in which to find and replace text, the text to find and
! replace, and the replacement text - all contained inside parentheses:

    replace$ (s$, "blind", "sexy")

! This prints the replaced text string returned by the function above:

    print s$

! The ftp.open function takes 4 parameters, the URL of an FTP server, the
! port to open, a username, and a password:

    ftp.open "ftp.site.com", 21, "username", "password"

! The ftp.put function takes 2 parameters, the name of a local file to
! tranfer to the remote FTP server, and the path/filename to create and
! send to on the server:

    ftp.put "temp.txt", "/public_html/temp.txt"

! The ftp.close function doesn't take a parameter.  It just closes the
! currently open FTP connection:

    ftp.close

! The graburl function takes 2 parameters, a string variable name in which
! to save the returned data from a specified URL, and a URL to read:

    graburl w$, "http://site.com/temp.txt"

! This prints the returned data read from the URL and stored in the
! variable created above:

    print w$

! The is_in function takes 2 parameters, a string to search for, and a
! string to search within.  It returns a value of 1 if the search string
! is found within the 2nd string, 0 if not found:

    q = is_in ("blind", "three blind mice")
```

```
! This line prints "It's in there" if the search string above is found
! (if the returned value is not 0):

    if q <> 0 then print "It's in there"

! The kb.show function shows Android's on-screen keyboard.  It does not
! take any parameters:

    kb.show

! The kb.hide function hides Android's on-screen keyboard.  It does not
! take any parameters:

    kb.hide

! The sqr() function takes 1 numeric parameter, between parentheses, and
! returns the square root.  The returned value can be assigned a variable
! name:

    s = sqr(16)
    print s

! This line does the exact same thing as the previous 2 lines.  It just
! prints the value of the sqr() function (the return value of the sqr()
! function is used as the parameter of the print function):

    print sqr(16)

! The array.load function creates a type of item list.  It takes 1 or more
! parameters, a variable name to assign to the list, and several items
! (in the case below, numbers), to add to the list:

    array.load p[], 0, 300, 200, 300, 200, 300

! The vibrate function takes 2 parameters, an array of durations to pause
! and vibrate, and a number to indicate whether that vibration pattern
! should be repeated (-1 indicates no repeat):

    vibrate p[], -1

! The gr.camera.manualshoot takes 1 parameter, a "pointer" number used to
! refer to the photo image taken:

    gr.camera.manualshoot p

! After the above function, look at the photo
! /sdcard/rfo-basic/data/image.jpg on your phone.  The results are stored
! in that temporary file (later in the tutorial, you'll learn how to use
! the returned pointer value to display and manipulate the image in
! graphic layouts).
```

The most important initial step in learning RFO Basic is learning how all the built-in commands/functions work, what their syntax is, and how they can be used *together* to perform complex operations and interactions with data. Progamming in general, and in fact all of computing, is all about processing data. Text input by the user, read from a file, or returned by a hardware sensor, for example, could potentially be used to determine what operation a program should perform next, or it could potentially contain a value which is plugged into a mathematical formula, or it could be used to move a graphic image to a given position on the screen, or it could be a block of text that needs to be searched and sorted for relavent content, etc. Text, images, sounds, etc. saved to files on your Android device or to FTP folders on a web server can be recalled and used later, edited, categorized, calculated upon, etc. as part of, for example, record keeping software, inventory, sales, and other data management systems, communications software, games, etc. Numbers, text, binary audio data, images and image manipulation parameters, video performance parameters, lists of related and categorized information about any real life topic, graphic coordinates and 3D movement computations in games, saved program settings, etc. - all of those things, and everything else that a computing device can interact with, is *data* of some sort, and all that data is processed by *functions*.

The list at http://laughton.com/basic/help/De_Re_BASIC!.htm#_Toc317902038 shows the format for all of RFO Basic's internal functions, also called it's "API". The rest of the document at http://laughton.com/basic/help/De_Re_BASIC!.htm explains in detail the input and output parameter(s) of each of those functions, and contains several dozen example programs which demonstrate their use.

The example programs from the document above are also installed automatically on your Android device, when you install the RFO Basic app. You can run any of them in RFO Basic by clicking the Android Menu button, then Load -> Sample_Programs(d). A complete list of all the built-in RFO Basic functions is also available by clicking Menu -> More -> Commands. Those two resources together provide essential documentation for the entire RFO Basic language, right on your Android device.

4.6 Concatenation

Concatenation is the joining together of pieces of text. In RFO Basic, you can concatenate text using the "+" symbol:

```
mystring$ = "Hello " + "World" + "!"
print mystring$
```

Concatenation is particularly useful when combining static text with data stored in variables or returned by functions. You will use code similar to the following in *virtually every program you write*:

```
name$ = input "What is your name?"
message$ = "Good to meet you " + name$ + "!"
popup message$, 0, 0, 1
```

Since you can't code an unkown user's name into your program, you must use a variable, and concatenate that text with other static (unchanging) text whenever you want to refer to the user's name in your program.

To concatenate text strings on multiple lines, use the tilde character (~):

```
print "all this text appears on " + ~
    "one line."
```

The "time" function sets all the following return variables to the current year, month, day, hour, minute, and second. Here's a nicely formatted concatened string that displays the current date and time:

```
time y$, m$, d$, h$, n$, s$

mytime$ = "The current date is " m$ + "-" d$ + "-" + y$ + ~
    ", and the time is " + h$, ":", n$, ":" s$

popup mytime$, 0, 0, 1
```

Again, since a programmer can't write the user's arbitrary current time into the code, variables and concatenation must be used to retrieve the current time, and refer to it by dynamically generated text.

4.7 Some Basic Text Formatting Guidelines

You can include quotes inside strings by prefacing them with the backslash character (\):

```
popup "\"quoted text\"",0,0,1
```

Multi-line text uses "\n" to represent a carriage return:

```
lines = "Line 1\nLine 2\nLine 3"
print lines
```

5. Conditions

Conditions are used to manage program flow ... to make "decisions" in your program. They are among the most important concepts in all types of programming.

5.1 If

The most basic conditional evaluation is "if":

```
if (this expression is true) then
    do this block of code
endif

! parentheses are NOT required
```

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

```
time y$, m$, d$, h$, n$, s$

h = val(h$)           % the val() function converts text data to a number.
                      % the variable "h" now stores that number.

if h > 12 then        % if the hour value is later than 12 o'clock, then...
    print "It's after noon."
endif
```

The above code could also be written as follows. The hour text is converted to a number, inline:

```
time y$, m$, d$, h$, n$, s$
if val(h$) > 12 then
    print "It's after noon."
endif
```

5.2 White Space and Indentation

RFO Basic *does not require line terminators* at the ends of code lines, and you can insert empty white space (tabs, spaces, etc.) as desired into code. It's standard practice in virtually all programming languages to indent code blocks which perform a group of related activities. For example, you could put multiple if-then conditions all on one line:

```
! Set some numeric variables (exp1 and exp2 now both equal the number 1):

exp1 = 1
exp2 = 1

! Perform 2 separate if evaluations using the variables, all on one line:

if (exp1 = 1) then (if (exp2 = 1) then (print "Yes"))
```

But it's much easier to understand the logic of each related code block above, if those conditional *blocks* are each indented. The following code does the exact same thing as the code above. It checks the second conditional evaluation **only if the first evaluation (if the variable exp1 = 1) is true**. So, *everything* indented inside that outside surrounding if-endif block only occurs if that first condition evaluates to true. It's much easier to follow the logic in this indented format:

```
exp1 = 1
exp2 = 1

if (exp1 = 1)
    if (exp2 = 1)       % These 3 lines only
        print "yes"     % run if the above
    endif               % evaluation is true
endif
```

NOTE: if you write an entire if-then condition on 1 line, then the word "then" is required. If you separate the "if" evaluation onto its own line, then the word "then" is *not required*. Notice that there are no "then" keywords in the code above.

5.3 If/Else

If/Else chooses between two blocks of code to evaluate, based on whether the given condition is true or false. Its syntax is:

```
if (condition)
    perform this code if the condition is true
else
    perform this code if the condition is false
endif
```

For example:

```
time y$, m$, d$, h$, n$, s$
if val(h$) >  8
    print "It's time to get up!"
else
    print "You can keep on sleeping."
endif
```

These types of evaluations are found in just about every type of computer program:

```
input "Username:", user$
```

```
if user$ = "validuser"
    print "Welcome back!"
else
    print "You are not a registered user."
endif
```

5.4 If/ElseIf/Else

You can choose between *multiple evaluations* of any complexity using the "If/ElseIf/Else" structure. If none of the cases evaluate to true, you can use "Else" to run some default code:

```
input "What is your name?", name$

if is_in ("a", name$) <> 0
    print "Your name contains the letter 'a'"
elseif is_in ("e", name$) <> 0
    print "Your name contains the letter 'e'"
elseif is_in ("i", name$) <> 0
    print "Your name contains the letter 'i'"
elseif is_in ("o", name$) <> 0
    print "Your name contains the letter 'o'"
elseif is_in ("u", name$) <> 0
    print "Your name contains the letter 'u'"
else
    print "Your name doesn't contain any vowels!"
endif
```

5.5 Multiple Conditions: "and", "or"

You can check for more than one condition to be true, using the "and" and "or" evaluations. 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 *both* responses are correct:

```
input "Username:", username$
input "Password:", password$

if ((username$ = "validuser") AND (password$ = "validpass"))
    print "Welcome back!"
else
    print "You are not registered!"
endif
```

This example responds favorably if the user chooses *any one* of three favorite colors:

```
input "What is your favorite color?", color$

if ((color$ = "blue") or (color$ = "red") or (color$ = "orange")
    print "That's one of my favorite colors too!"
else
    print "We don't like any of the same colors :("
endif
```

5.6 Switch

This evaluation allows you to compare as many values as you want against a main value, and run a selected block of code for any matching value, with an optional default block of code to be run if no matches are found:

```
input "What's your favorite day of the week?", favoriteDay$

sw. begin favoriteDay$
sw.case "Monday"
    print "Monday is the worst!  The work week begins..."
sw.break
sw.case "Tuesday"
sw.case "Thursday"
    print "Tuesdays and Thursdays are both ok, I guess..."
sw.break
sw.case "Wednesday"
    print "The hump day - the week is halfway over!"
sw.break
sw.case "Friday"
    print "Yay!  TGIF!"
sw.break
sw.case "Saturday"
sw.case "Sunday"
    print "Of course, the weekend!"
sw.break
sw.default
    print "You didn't type in the name of a day!"
sw.end

! note that you can perform the same action for 2 or more conditions
! by placing those conditions consecutively before an sw.break
! (as in the Tuesday/Thursday and Saturday/Sunday conditions above)
```

The conditional structures you've seen here will help your programs perform appropriate actions in RFO Basic, based upon user input, data content, and other potentially changing situations.

6. More Program Flow Structures

6.1 Gosub/Return

You can label any part of your program, jump to that section of code, and then return back to the original jumping point using the "Gosub" and "Return" commands. Use a colon (:) to label the portion of code to jump to:

```
print "The program has begun."
gosub whee
gosub whee
gosub whee
print "The program has ended"
end

whee:
print "Whee!"
return
```

This sort of jumping around code is typically considered depricated, and is typically replaced by function structures, but they can be useful for creating simple subroutines.

6.2 Goto

"Goto" also allows you to jump to a labeled portion of code, but does not return. It is especially useful for restarting an entire program. For example, when a game is completed, you could provide an option to rerun the entire program, just by labeling the beginning of the program with "restart:". The following program will run endlessly until stopped (using the back key on your Android device):

```
restart:
print "Ahhhhh!!!"
goto restart
```

6.3 Creating Your Own Functions

You can create your own functions to process data. Use the following format:

```
fn.def <function_name> (<data parameter(s), inside parentheses>)
    commands to process the parameter data
    fn.rtn <data_to_return>
fn.end
```

Here's an example:

```
fn.def average(x,y,z)
    avg = (x + y + z) / 3
    fn.rtn avg
fn.end

print average(4,7,2)
print average(24,56,14)

a = average(83, 2, 71)
print a
```

Use functions to avoid duplicating code that performs the same action(s), or to process data the same way, in multiple locations in your code.

6.3.1 Global and Local Variables in Functions

By default, values used inside functions are treated as *local*, which means that if any variables are changed inside a function, they will NOT be changed throughout the rest of your program:

```
x = 10

fn.def changex(x)
    x = x + x
    print "X = " + x
fn.end

changex(q)   % prints 'X = 20'

print "Outside the function, X is still: " + x   % still 10
```

You can change this default behavior, and specify that any parameter value be treated as *global* (CHANGED throughout the rest of your program), by using "&" symbol on the global parameter. This is referred to as "passing the variable by reference":

```
x = 10

changex(&x)      % inside the function, x is now 20, and it
                 % has been CHANGED outside the function too

print "Outside the function, X is now changed to: " + x
```

In most cases, to obtain a value computed within a function definition, you'll simply return a value, instead of passing global variables. For example, it's much easier to just return a computed value from the above function, and use the returned value elsewhere in your program:

```
x = 10

fn.def addedval(x)
    y = x + x
fn.end

y = addedval(x)
print y
```

7. Data Structures: Arrays, Lists, Bundles, and Stacks

In RFO Basic, *multiple pieces of grouped data items* are stored in "arrays", "lists", "bundles", and "stacks". These data structures are basically used to store *lists* of data. Lists of data are tremendously important in virtually all types of app programming.

7.1 Arrays

Arrays are most typically used to store lists of info which *don't change in length*. To create an array in RFO Basic, use the "array.load" function. The parameters of the array.load function are a list of strings *or* numbers, separated by commas - those two types of data can NOT be mixed in arrays (an array must contain either all numbers, or all text items). The first item in the parameter list is the variable label chosen to represent the items in the list:

```
array.load nums[],1,3,5,2+5,9
```

You can select items from the array using a numerical "index". Indexes in RFO Basic are "one based", meaning the item count starts with 1 (many programming languages are zero based):

```
print nums[1]   % prints 1 - the first item in the array
print nums[2]   % prints 3 - the second item
print nums[4]   % prints 7 - the fourth item
```

You can continue written data onto new lines of code using the tilde (~) character:

```
array.load days$[], "Monday", "Tuesday" ~
    "Wednesday", "Thursday" ~
    "Friday", "Saturday", "Sunday"
print days$[1]
print days$[6]
```

Arrays can only be created ("dimensioned") once during a program. If you need to reload an array with different values, first use the "array.delete arrayname$[]" function, then re-create the array. It's a good idea to get in the habit of using array.delete every time you create an array:

```
array.delete a$[]
array.load a$[], "adf", "qwe"
```

RFO Basic contains a number of functions to organize and manage data in arrays. Try typing or pasting all these functions into the RFO Basic interpreter, to see how they work:

```
array.reverse nums[]
print nums[]

array.shuffle nums[]
print nums[]

array.sort nums[]
print nums[]

array.copy nums[], newnums[]
print newnums []

array.length lngth, nums[]
print lngth

array.sum s, nums[]
print s

array.average a, nums[]
print a

array.min m, nums[]
print m

array.max x, nums[]
print x

array.variance v, nums[]
print v

array.std_dev sd, nums[]
print sd

array.delete nums[]
print nums[]
```

Multidimentional arrays allow for lists of lists:

```
dim l,[1,2,3],[11,12,13],[21,22,23]
print l[1][1]
print l[2][1]
print l[3][2]
```

Multidimentional arrays are useful for storing matrices of coordinate locations in graphic applications, and other situations in which groups of related lists of info need to be placed into a larger single list.

IMPORTANT: The size of an array cannot be changed once it is created. If you need to add a new item to an array, beyond the array's original size, you must first make a copy of it, then delete the orginal array, then create a new array of the desired size, and then populate it with data from the copied array, and then add the new item(s) to it. Arrays are designed to store lists of info that don't change, such as the list of months in a year, the top ten high scores in a game, the locations of a given number of unchanging images in a graphic layout, etc.

7.2 Lists

Lists work much like arrays, but their *size can be changed* as needed. Lists are designed to hold and manage user data, such as inventory items in a retail store, the names of friends in a contact database, the list of files in a directory, etc. Like arrays, lists can only contain a single type of data. All the data in any list must be either all strings (text), or all numbers. If you need to store mixed data types, store each element as a string, and then use the val() function to convert numerical values.

Lists are created using the "list.create" function. The first parameter tells whether the list will contain text ("string") or number items. The second parameter is a label for the list (a numerical "pointer" variable):

```
list.create s, myfriends
list.create n, mynumbers
```

The "list.add" function adds items to the list. The first parameter is the label (as entered in the list.create function above). Notice that the tilde symbol (~) is used to continue items onto additional lines of code:

```
list.add myfriends, "John", "Bob", "Mike", "Joe" ~
    "Sue", "Bill", "Amanda", "Jim", "George", "Tim"
```

RFO Basic has a number of functions that allow you to organize and manage data in lists. Try these examples in the RFO Basic interpreter:

```
list.size myfriends, s
print s

list.type myfriends, t$
print t$

list.insert myfriends, 2, "Steve"
list.size myfriends, s
print s

list.remove myfriends, 3
list.size myfriends, s
print s

list.get myfriends, 3, s$
print s$

list.replace myfriends, 5, "Paul"

list.search myfriends, "Paul", c
print c

list.add.array myfriends, days$[]
list.size myfriends, s
```

```
print s

list.create s, newfriends
list.add newfriends, "Tristan", "Peter"
list.add.list myfriends, newfriends
list.size myfriends, s
print s

list.ToArray myfriends, myfriends$[]
print myfriends$[7]

list.clear myfriends
list.size myfriends, s
print s
```

Lists are the data storage workhorse of RFO basic. You'll use them regulary to manage all types of data collections in your programs.

7.3 Bundles

Bundles allow you to save pieces of data in a list, with an associated "key", or label, so that you can look up the associated data later by name. You could, for example, save invoice numbers as a key, and clients' names as the associated data in a bundle. You could then keep all of the client's account information in a separate list structure. This could, for example, provide a data structure to save and look up contact information for any account transaction handled by a business. Bundles can contain mixed numeric and text values. Here are the functions you can use to create and manage bundles:

```
bundle.create n

bundle.put n, "19327", "Frank Thompson"

bundle.get n, "19327", s$
print s$

bundle.keys n, mykeys
list.get mykeys, 1, k$
print k$

bundle.type n, "19327", t$
print t$

bundle.clear n
```

8. Loops and Structured Data

"Loop" structures provide ways to methodically repeat actions, manage program flow, and automate lengthy data processing activities. Loops are often used to *step through the individual items in a list of data*.

8.1 While/Repeat and Do/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
repeat
```

This example counts to 5:

```
x = 1    %create an initial counter value

while x <= 5
    print x
    x = x + 1
repeat
```

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
Repeat
```

"Incrementing" counter values, as in the example above, is an extremely common technique in all types of programming. You'll see examples of it regularly in code, whenever you want to keep a count of items.

You can use While/Repeat to create forever repeating loops. Simply use an evaluation that is always true, and use the "wr.break" function to end the loop. Notice how the While loop and If evaluations are indented to clearly separate the logic:

```
print "Please wait 5 seconds ...",0,0,1
alarmtime = clock() + 5000

while 1 = 1                                % this is always true
    pause 1000                             % wait 1 second
    if clock() = alarmtime
        print "5 seconds have passed"      % these lines are only executed if
        w_r.break                          % the clock time = the alarm time
    endif
repeat
```

Here's a more interactive version using some info provided by the user:

```
input "What do you want to be reminded of?", eventname$
input "Seconds to wait:", seconds
endtime = clock() * 1000 + seconds
time y$, m$, d$, h$, n$, s$
mytime$ = h$ + ":" + n$ + ":" + s$
print "It's now " + mytime + ", and you'll be alerted in "
    + seconds + " seconds."
while 1 = 1
    pause 1000
    if clock() = endtime
        print "It's now " + endtime + ", and " + seconds +
            " seconds have passed.  It's time for: " + eventname$
        w_r.break
    endif
repeat
```

"Do-Until" loops operate similarly to while-repeat loops, except they ensure that *at least one iteration* of the statements inside the loop is executed:

```
i = 1
do
    print i
until i = 1
```

This type of loop is often used to wait for keystrokes from the user:

```
do

    ! This function gets the current keystroke:

    inkey$ k$

    ! The "@" symbol is used to represent no key being pressed:

    if ((k$ <> "@") & (k$ <> "key 4")) then let done = 1

until done   % if something other than no key, or key 4 is pressed, stop
print "You pressed the '" + k$ + " key."
```

Memorize the code above. You'll use it whenever you want to wait for a single keystroke from the user.

8.2 Looping Through Lists of Data: FOR

"FOR/Next" loops simply count through a range of numbers:

```
for i = 0 to 6
    print i
next
```

You can use conditional tests within a FOR loop, to perform different operations on each numbered item:

```
for i = 1 to 100
    if ((mod(1,3) = 0) and (mod(i,5) = 0))   % mod checks the remainder
        print "fizzbuzz"                     % of a division operation
    elseif mod(i,3) = 0
        print "fizz"
    elseif mod(i,5) = 0
        print "buzz"
    else
        print i
    endif
next i
```

You can use the sequential counting ability of FOR loops **to pick sequential items from an array or a list *using numbered indexes***:

```
! This creates a new list, labeled by the variable "months":

list.create s, months
list.add months, "January", "February", "March", "April", "May", "June"~
    "July", "August", "September", "October", "November", "December"

! This gets the number of elements in the months list, and stores that
! value in the variable "size"

list.size months, size

! This loop counts from 1 to the number of items in the list, and does
! something with each item in the list:

for i = 1 to size

    ! This picks the numbered index item from the list, and assigns that
    ! item to the variable "m$":

    list.get months, i, m$

    ! This prints some concatenated text, using the string above:

    print "Month " + str$(i) + ":  " + m$

next i
```

The "Step" option of a FOR loop allows you to *skip* a given number of items, each time through the loop:

```
for i = 0 to 12 step 3       % skip to every third number
    print i
next i
```

And to count backwards:

```
for i = 5 to 1 step -1       % start on 5, end on 1, subtract 1
    print i                  % each time through the loop
next i
print "Blast off!"
```

"Step" is especially useful because it allows you to ***skip*** *items in an array or list*:

```
list.create s, months
list.add months, "January", "February", "March", "April", "May", "June"~
    "July", "August", "September", "October", "November", "December"

list.size months, size

! Count from 1 to the last item in the list, skipping by 3:

for i = 1 to size step 3
    list.get months, i, m$                        % pick every 3rd item
    print "Month " + str$(i) + ":  " + m$         % and print it
next i
```

This technique is especially useful when creating and using lists of structured data. For example, the following list contains three fields of data for each person (name, address, and phone), in consecutive order. Pay particular attention to how the FOR/Next/Step loop is used to pick out blocks of 3 consecutive pieces of data from the list:

```
list.create s, myusers
list.add myusers,"John Smith", "123 Tine Ln. 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"

print "All users:"
print ""

list.size myusers, size

for i = 1 to size step 3

  ! Pick and print the item at the current index  (every 3rd field, name)

  list.get myusers, i, n$
  print "Name:     " + n$

  ! Pick and print the item at the next index (current index + 1, address)

  list.get myusers, i+1, a$
  print "Address:  " + a$

  ! Pick and print the item 2 after the current index (index + 2, phone)

  list.get myusers, i+2, p$
  print "Phone:    " + p$
  print ""

next i
```

The code pattern above is *very* useful when creating data management apps. Little data lists such as the one above can hold all sorts of useful related blocks of data (phone books, recipes, etc.). You can group different fields of data sequentially, and read any selected field of information from any record using a FOR/Next/Step loop.

You can also use FOR loops to *alter* specific fields of data in a list. The following example checks *each phone number field* in the myusers list (every third item in the list, starting with the third item. If any phone field is empty, it changes that field to "000-000-0000":

```
for i = 3 to size step 3
    list.get myusers, i, m$
    if m$ = ""
        list.replace myusers, i, "000-000-0000"
    endif
next i
```

8.3 Choosing Items From Lists - The "Select" Function

RFO Basic has a built in "select" function to allow users to easily choose from items in a list. It takes 3 parameters:

1) A *return* parameter that holds the *index number* of the *item selected by the user*

2) The variable label that refers to the list

3) A text message to display to the user. The text message is shown on top of the select screen, and also displayed quickly in a popup box. If the text message is left blank (the empty quotes string ""), then no popup message occurs:

```
list.create s, months
list.add months, "January", "February", "March", "April", "May", "June"~
    "July", "August", "September", "October", "November", "December"

! Allow the user to select an item from the list:

select selecteditem, months, ""

! Pick out and print the selected item:

list.get months, selecteditem, m$
print m$
```

The select function can also be used to choose items from an array:

```
array.load months$[], "January", "February", "March", "April", "May"~
  "June", "July", "August", "September", "October", "November", "December"

select selecteditem, months$[], ""
print months$[selecteditem]
```

There is an optional fourth parameter that you can provide the select function, which determines if the user selected the item with a short click or a long hold. This option can be useful for doing *different* things with the selected data, depending upon how the user makes the physical selection:

```
array.load months$[], "January", "February", "March", "April", "May"~
  "June", "July", "August", "September", "October", "November", "December"
m$ = "Click an Item to Print, Click and HOLD to DELETE"
select selecteditem, months$[], m$, d
print months$[selecteditem]

! The final optional parameter in the select function determines if the
! user clicked short or held long on the selected item.
! 0 = short, 1 = long

if d = 1 then print "Item will be deleted"
```

8.4 Selecting Records From Structured Lists Data

You've already seen how to use a FOR/Next/Step loop to select specified fields of items from a list of structured data (in the following case, every 3 items from the list are a name, address, and phone number):

```
list.create s, myusers
list.add myusers,"John Smith", "123 Tine Ln. 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"
list.size myusers, size
for i = 1 to size step 3
  list.get myusers, i, n$
  print "Name:     " + n$
next i
```

You can use this technique to allow users to select from a specific *field* in a record (i.e., in this case, either name, address, or phone number of a user), and then do *something useful* with the selected data. In this example, the user can select a name from the user data, and then the name, address, and phone number for just that user is displayed:

```
list.create s, myusers
list.add myusers,"John Smith", "123 Tine Ln. 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"
list.size myusers, size

! Create a NEW list containing just the NAMES (using a for loop to
! pick out every third item in the list above):

list.create s, namelist
for i = 1 to size step 3
  list.get myusers, i, n$
  list.add namelist, n$
next i

! Label this section of code, so we can jump back later:

selectname:

! Allow the user to select a name from the new names list:

select s, namelist, "Click and HOLD Any Name to End the Program", d
if d = 1 then end

! Print the selected record:

cls                          % clear screen
list.get myusers, s, n$      % the selected item (name)
print "Name:     "
print n$
list.get myusers, s+1, a$    % the next item, in order (address after name)
print "Address:  "
print a$
list.get myusers, s+2, p$    % the next item, in order (phone after address)
print "Phone:    "
print p$

! The following loop simply waits for the user to press a key,
! then returns to the label above:

do
    inkey$ k$
    if ((k$ <> "@") & (k$ <> "key 4")) then let done = 1
until done
goto selectname
```

Get to know the above example well. Experiment with it and commit it to memory. You will find the general concept and code pattern very useful in many situations. You can use it to create common types of data retrieval apps: recipe collections, address books, inventory information lists, etc.

8.5 Saving Lists of Structured Data to a Storage Medium - Serialization

One of the most important things that computing devices allow users to do is save and retrieve data from local and remote storage mediums. To save lists of text data in RFO basic, a useful technique is to "serialize" the info as a long piece of text, with each item in the list separated by a specified text character. To do this, use a FOR loop to cycle through each item in the list, and concatenate each of the items together, with the specified character in between each item in the list. Then save the concatenated text to a file:

```
list.create s, myusers
list.add myusers,"John Smith", "123 Tine Ln. 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"
list.size myusers, size

! Concatenate each item in the "myusers" list together into one long
! string, with each item separated 3 pound characters (#):

savedata$ = ""                      % create a new string variable

for i = 1 to size                   % loop through every item
    list.get myusers, i, m$

    ! Add 3 pound symbols after every item in the list, EXCEPT the last
    ! item:

    if i = size
        savedata$ = savedata$ + m$          % concatenate, if last item
    else
        savedata$ = savedata$ + m$ + "###"  % concatenate every other item
    endif
next i

! All that concatenated text now exists in the string labeled "savedata$".
!  Now simply write that string to a file:

text.open w, myfile, "myusers.txt"
text.writeln myfile, savedata$
text.close myfile
```

To retrieve the data and convert it back to an RFO Basic list structure, follow this pattern:

```
! First, read the saved text file:

text.open r, myfile, "myusers.txt"
text.readln myfile, serialdata$
text.close myfile

! Use the "split" function to separate each stored record from the
! loaded text.  The character we used to separate those items was
! the pound symbols (###).  The split function will separate each
! item at those specified pound characters, and create an array
! holding each of the separated items:

split myusers$[], serialdata$, "###"

! Finally, convert the array structure to a list.  To do that,
! create a new list and add the items from the array created above,
! using the list.add.array function (basically this converts an
! array to a list, so that the resizing ability of the list
! structure can be utilized):

list.create s, myusers
list.add.array myusers$[]

! Now you can use the list as usual, loop through it, print selected
! items, edit specified items, etc.:

list.size myusers, size
for i = 1 to size
    list.get myusers, i, m$
    print m$
next i
```

Serializing data ensures that even multiline text in lists is separated properly and that each entry is stored as an individual item within a monolithic chunk of text. Just be sure that the character(s) you use to separate each data item would never appear in your user data. For example, if your user data fields above were ever to contain 3 pound characters in the example, the split function would incorrecly split the saved text at that point, as separate records. RFO Basic allows the splitting character(s) to be arbitrarily complex. You could, for example, use "!z%l3#a@q_x;" as the split string, which has virtually no likelihood of occuring naturally in your user text.

NOTE: The split function actually accepts a "regex expression" as the split string. Avoid using the single characters ()[]{}.|$^*+? to separate data because those characters have special meaning when creating matching patterns. Regex patterns are a powerful tool for searching text patterns. Lookup "regex tutorial" in Google to learn how they work.

8.6 Sending Serialized Data to a Web Server

Serialization allows you to transfer lists of data to other computing platforms and storage mediums, such as web servers, as *single text files*. You can use RFO Basic's FTP commands to save the serialized data to a remote server, for example:

```
list.create s, myusers
list.add myusers,"John Smith", "123 Tine Ln. 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"
list.size myusers, size
savedata$ = ""
for i = 1 to size
    list.get myusers, i, m$
    savedata$ = savedata$ + m$ + "###"
next i
text.open w, myfile, "myusers.txt"
text.writeln myfile, savedata$
text.close myfile

! The above code is exactly the same as the previous example.
! Now just upload the serialized text file to your server:

ftp.open "ftp.site.com", 21, "username", "password"
ftp.put "myusers.txt", "/public_html/myusers.txt"
ftp.close
```

The following code will read data from the above uploaded file on your web server, and import it into a list structure on a local Android device. This provides a very simple method of sharing useful lists of data between users connected via the Internet. You can use it to backup up local data, transfer data to different devices, etc.:

```
graburl serialdata$, "http://site.com/myusers.txt"
split myusers$[], serialdata$, "###"
list.create s, myusers
list.add.array myusers$[]

! Done.  Now do something with the list:

list.size myusers, size
for i = 1 to size
    list.get myusers, i, m$
    print m$
next i
```

8.7 Sharing data between different programming languages and environments

Any useful modern programming language will provide functions to split character delimited text files into lists of data. The tutorial at http://re-bol.com/rebol.html#section-9.3, for example, explains how to separate such data using the REBOL language. You could use the following code, for example, to make use of the uploaded data in a REBOL CGI web server application. This code reads the uploaded data from the previous example, and prints it out into the user's web browser, line by line:

```
myusers: parse read %myusers.txt "#"
foreach item myusers [
    print item
    print "<br>"
]
```

Learning to use other programming languages to manage data becomes easier, once you see that every language has its own way of labeling variable data, using functions to perform actions, evaluating conditions, looping through list structures, reading and writing file I/O, etc. The syntax is typically different, but the methodology for accomplishing similar goals generally involves the same types of logic, and comparable coding structures. The data you create and/or process in RFO Basic can typically be transferred and used in other environments very easily, and visa-versa.

8.8 Using the Select Function to Create Menus

You can use the Select function to accept user choices regarding program control. This provides a very simple and easy-to-use menuing system:

```
array.load choices$[], "Option 1", "Option 2", "Option 3"
select s, choices$[], ""
if choice$[s] = "Option 1"
    ! code to run for choice 1:
    popup "You chose option 1",0,0,1
elseif choice$[s] = "Option 2"
    ! code to run for choice 2:
    popup "You chose option 2",0,0,1
else
    ! code to run for last choice:
    popup "You chose option 3",0,0,1
endif
```

You can create submenus by simply running another select function, *inside* the If/Then/Else responses of the main menu:

```
array.load mainmenu$[], "Option 1", "Option 2", "Option 3"
array.load submenu1$[], "Sub-Option1-1", "Sub-Option1-2"
array.load submenu2$[], "Sub-Option2-1", "Sub-Option2-2"
```

```
select s, mainmenu$[], ""
if mainmenu$[s] = "Option 1"
    select s1, submenu1$[], ""
    if submenu1$[s1] = "Sub-Option1-1"
        popup "You chose option 1, sub-option 1",0,0,1
    else
        popup "You chose option 1, sub-option 2",0,0,1
    endif
elseif mainmenu$[s] = "Option 2"
    select s2, submenu2$[], ""
    if submenu2$[s2] = "Sub-Option2-1"
        popup "You chose option 2, sub-option 1",0,0,1
    else
        popup "You chose option 2, sub-option 2",0,0,1
    endif
else
    popup "You chose option 3",0,0,1
endif
```

You can use the select function for simple Yes/No/Cancel operations:

```
array.load choices$[], "Yes", "No", "Cancel"
select s, choices$[], "Would you like to continue?"
if choices$[s] = "Yes"
    popup "Running...",0,0,1
else
    popup "Cancelling...",0,0,1
endif
```

Learning to use the "select" function to accept menu choices, along with the "input" and "text.input" functions to acquire text typed by users is very useful. Together, those tools provide amazingly powerful and easy ways to control program flow and to collect user data input. GUI (graphic) windows and widgets *are* possible in RFO Basic, but in most cases, within the simple and specialized types of apps that you'll create in the Android environment, you'll often need nothing more complicated than menus and text input boxes. Learning to provide data input and to manage interactivity using these simple tools will go a *long way* towards making you a capable RFO Basic coder.

9. COMPLETE EXAMPLE APPS - Learning How All The Pieces Fit Together

The examples in this section demonstrate how RFO Basic code is put together to create complete programs. The code is heavily commented to provide line-by-line explanations of how each element works. Downloadable executable apps (.apk file) screen shots of these examples are available at:

```
http://rfobasic.com/examples  (under construction)
```

9.1 Math Test

This program helps students test their math addition skills. It allows a user to choose upper limits for each of the 2 numbers in the math test questions, then repeatedly displays random addition questions with booth addends within 0 and the limit range. The "format$" and "replace$" functions are used to display neatly formatted questions, and a total count of correct and incorrect answers is tallied.

```
! First, use the "input" function to request upper limit numbers from the.
! user.  Save the response in the variables "l1" and "l2".  Notice that
! there are no "$" characters in these variables, because they are meant
! to hold numeric values:

input "High limit of the first number", l1
input "High limit of the second number", l2

! These next two variables are used to count the number of correct and
! incorrect answers entered by the user.  The count for both is initially
! set to zero:

c = 0
i = 0

! Label this point in the program to jump back later:

start:

! Next, generate 2 random numbers.  The rnd() function generates a number
! between 0 and 1.  That's multiplied by an upper limit number (entered
! earlier by the user), and then rounded to the nearest whole number,
! using the round() function.  Notice that the entire "rnd() * l1"
! expression is treated as the argument for the round() function.  The
! results are stored in the variables "n1" and "n2":

    n1 = round(rnd() * l1)
    n2 = round(rnd() * l2)

! Because the rounded number above is a decimal number, we need to convert
! it to a string, to be concatenated and displayed to the user in the form
! of a question.  We could use the str$() function, but that would display
! each number with a decimal point and a trailing 0 (i.e. "7.0").  To make
! the display a little nicer, we'll use the format$() function to display
! up to 5 characters, with no trailing decimal characters.  Save the
! result in the variable n1$ (notice the use of the "$" character, because
! this variable holds a string):

    n1$ = format$("#####", n1)

! Trim any trailing spaces from the second number, using the replace$()
! function.  Save the result in the variable n2$:

    n2$ = replace$(format$("#####", n2), " ", "")

! Concatenate the numbers above into a nicely formatted question, and use
! the input function to request an answer from the user.  Save the user's
! response in the variable "answer".

    input n1$ + " + " + n2$ + " = ", answer

! If the user's answer is correct (the total of n1 + n2), add 1 to the
! count of correct answers (stored in the numeric variable "c"), and
! display a positive message to the user.  If the answer is incorrect,
! incorrect the counter variable that holds the number of incorrect
! answers ("i"), and display the appropriate message:

    if answer = n1 + n2
        c = c + 1
        popup "CORRECT!", 0, 0 ,0
    else
        i = i + 1
        popup "Incorrect", 0, 0 ,0
    endif
```

```
! Wait 2 seconds for the user to see the message:

    pause 2000

! Go back the to "start" label and do the question routine all again:

goto start

! If at any point the back button is clicked duing an input operation, an
! error is raised, and the program automatically jumps to the label marked
! "OnError":

OnError:

! When the OnError event occurs, display a message that concatenates the
! total number of correct and incorrect answers (you could use the
! format$() function here to display these numbers without trailing
! decimal characters, if desired):

    popup str$(c) + " correct, " + str$(i) + " incorrect.",0,0,0
```

Here's the whole program, without comments:

```
input "High limit of the first number", l1
input "High limit of the second number", l2
c = 0
i = 0
start:
    n1 = round(rnd() * l1)
    n2 = round(rnd() * l2)
    n1$ = format$("#####", n1)
    n2$ = replace$(format$("#####", n2), " ", "")
    input n1$ + " + " + n2$ + " = ", answer
    if answer = n1 + n2
        c = c + 1
        popup "CORRECT!", 0, 0 ,0
    else
        i = i + 1
        popup "Incorrect", 0, 0 ,0
    endif
    pause 2000
goto start
OnError:
    popup str$(c) + " correct, " + str$(i) + " incorrect.",0,0,0
```

9.2 Text Editor App

Here is a full text editor app that allows you to choose a file from your Android storage card, edit it, and then save it back to the file system when done.

```
! The following flag variable will be used later to determine if the user
! has already entered a starting folder:

flag = 0

! This label marks a point in the code which can be jumped back to later:
start:
```

```
! After the program runs once, the flag variable gets set to 1 and the
! startfolder$ variable gets choosen by the user.  If this is the first
! time through the program loop ("flag" variable still equals 0), then a
! default starting folder needs to be set:

if flag = 0 then startfolder$ = "/sdcard/rfo-basic/data/"

! This line asks the user for a folder, with the default folder displayed.
! If the startfolder$ variable has been set on a previous loop through the
! program, that value is used as the default:

input "Folder", folder$, startfolder$

! Save the chosen folder in the startfolder$ variable, and set the flag
! variable to 1 (to indicate that a starting folder has now been chosen by
! the user):

startfolder$ = folder$
flag = 1

! Remember, the default data folder in RFO Basic is
! /sdcard/rfo-basic/data/
! The characters "../" are used to go up one folder level.  So,
! for example, "../sdcard/rfo-basic/data/" = "/sdcard/rfo-basic/", and
! "../../../sdcard/rfo-basic/data/ is the same as the top level (root)
! directory of your Android file system.  In order to refer to the folder
! chosen by the user, we need to refer to that folder in terms relative
! to RFO Basic's default folder.  This line concatenates a string to
! properly refer to the relative location of the chosen folder:

rfofolder$ = "../../../" + folder$

! This line deletes the array "files$[]".  If the array is not deleted,
! the next line of code will produce an error when the program loops
! back to the beginning (remember, arrays can't be re-dimensioned without
! being deleted):

array.delete files$[]

! This line reads the directory listing of the chosen folder, and saves
! that listing in the array variable "files$[]":

file.dir rfofolder$, files$[]

! This line sorts the array alphabetically:

array.sort files$[]

! This line allows the user to choose a file name from the sorted file
! list:

select s, files$[], ""

! This line concatenates the chosen folder together with the selected
! file, to create a full path to the desired file:

myfile$ = rfofolder$ + files$[s]

! This line reads the contents of the file, and saves the read text into
! the string variable "unedited$":

grabfile unedited$, myfile$
```

```
! This line allows the user to edit the read text, and saves the edited
! content into the string variable "edited$":

text.input edited$, unedited$

! This line opens a text file for writing, and creates the pointer
! variable "filename" to refer to the file:

text.open w, filename, myfile$

! This line writes the contents of the "edited$" variable to the file:

text.writeln filename, edited$

! This line closes the file:

text.close filename

! This line alerts the user with a short message:

popup "Saved",0,0,1

! This line jumps back to the label at the beginning of the code, and
! starts the whole process over again, to choose and edit another file:

goto start
```

Here's the entire program, without comments - it's short and simple:

```
flag = 0
start:
    if flag = 0 then startfolder$ = "/sdcard/rfo-basic/data/"
    input "Folder", folder$, startfolder$
    startfolder$ = folder$
    flag = 1
    rfofolder$ = "../../../" + folder$
    array.delete files$[]
    file.dir rfofolder$, files$[]
    array.sort files$[]
    select s, files$[], ""
    myfile$ = rfofolder$ + files$[s]
    grabfile unedited$, myfile$
    text.input edited$, unedited$
    text.open w, filename, myfile$
    text.writeln filename, edited$
    text.close filename
    popup "Saved",0,0,1
goto start
```

9.3 Web Page Editor

This program is a variation on the simple text editor above, which allows you to edit HTML files (or any other text files) on your web server:

```
! Set a flag variable to determine if the user has entered FTP login info:

flag = 0
```

```
! Mark a point in the code to repeat back to later:

start:

! If this is the first time running the program loop, set some default
! FTP values:

if flag = 0
    startsite$ = "ftp.site.com"
    startfolder$ = "/public_html/"
    startusername$ = "user"
    startpassword$ = "pass"
endif

! Request FTP value, using either the default values above, or the values
! entered by the user on a previous loop through the code:

input "FTP URL", site$, startsite$
input "File Path", folder$, startfolder$
input "User Name", username$, startusername$
input "Password", password$, startpassword$

! Set the flag variable to indicate that FTP values have been entered by
! the user, and save those values for later:

flag = 1
startsite$ = site$
startfolder$ = folder$
startusername$ = username$
startpassword$ = password$

! Open the FTP connection, and download the names of the files in the
! selected directory:

ftp.open site$, 21, username$, password$
ftp.dir filelist

! Add a choice to the list of file names, to create a new file:

list.add filelist, "New File..."

! Allow the user to select a file:

select indx, filelist, "Select a file:"
list.get filelist, indx, file$

! If the user selected "New File...", allow them to enter the name of the
! new file to be created on the server, and then empty the contents of the
! local temp file.  If the user selected the name of an existing file,
! save the contents of that file to the local temp file ("temp.txt"):

if file$ = "New File..."
    input "File Name:", file$, "file.html"
    Text.open w, tempfile, "temp.txt"
    Text.writeln tempfile, ""
    Text.close tempfile
else
    ftp.get file$, "temp.txt"
endif

! Read the contents of the temp file in the variable "unedited":
```

```
grabfile unedited$, "temp.txt"

! Allow the user to edit the above text, and save the edited text in the
! variable "edited$":

text.input edited$, unedited$

! Save the edited text to the temp file:

text.open w, filename, "temp.txt"
text.writeln filename, edited$
text.close filename

! Upload to the contents of the temp file back to the web server, and
! alert the user when done:

ftp.put "temp.txt", file$
ftp.close
popup "Saved",0,0,1

! Run the whole program again:

goto start
```

Here's the whole program, without comments:

```
flag = 0
start:
    if flag = 0
        startsite$ = "ftp.site.com"
        startfolder$ = "/public_html/"
        startusername$ = "user"
        startpassword$ = "pass"
    endif
    input "FTP URL", site$, startsite$
    input "File Path", folder$, startfolder$
    input "User Name", username$, startusername$
    input "Password", password$, startpassword$
    flag = 1
    startsite$ = site$
    startfolder$ = folder$
    startusername$ = username$
    startpassword$ = password$
    ftp.open site$, 21, username$, password$
    ftp.dir filelist
    list.add filelist, "New File..."
    select indx, filelist, "Select a file:"
    list.get filelist, indx, file$
    if file$ = "New File..."
        input "File Name:", file$, "file.html"
        Text.open w, tempfile, "temp.txt"
        Text.writeln tempfile, ""
        Text.close tempfile
    else
        ftp.get file$, "temp.txt"
    endif
    grabfile unedited$, "temp.txt"
    text.input edited$, unedited$
    text.open w, filename, "temp.txt"
    text.writeln filename, edited$
    text.close filename
    ftp.put "temp.txt", file$
    ftp.close
    popup "Saved",0,0,1
goto start
```

9.4 Internet Chat Room

This example is a chat application that allows a collective group of users to send instant text messages back and forth across the Internet. 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).

```
! First, ask the user for the URL of an FTP server that will hold the text
! of the chat room conversation:

input "FTP URL", site$, "ftp.site.com"                  % default URL

! Next, get the path name of the folder/directory that will hold the text
! file containing the conversation:

input "Folder", folder$, "./public_html/"               % default folder

! Next, get the name of the text file name that will hold the chat text:

input "File", file$, "chat.txt"                         % default file

! Next, get the FTP username:

input "User Name", username$, "user"

! Get the FTP password:

input "Password", password$, "pass"

! Get the user's screen name:

input "Your Screen Name", name$, "Name"

! Open the FTP connection:

ftp.open site$, 21, username$, password$

! Change directory to the necessary folder:

ftp.cd folder$

! This portion of code checks to see if the file$ text file already exists
! on the server.  First, set a flag variable to indicate that by default,
! it's assumed the file doesn't exist:

exists = 0

! Now get a listing of files in the current FTP folder:

ftp.dir files

! Loop through the files in the folder list to see if any of the existing
! file names match the file name given above.  If a match is found, change
! the "exists" flag variable to equal 1:

list.size files, s
for i = 1 to s
    list.get files, i, page$
    if file$ = page$ then exists = 1
next i

! If the file doesn't exist (i.e., the "exists" flag variable hasn't been
! changed from 0 to 1), then create a new, blank text file and transfer
! it to the server:

if exists = 0
    console.save "temp.txt"              % create new blank text file
    ftp.put "temp.txt", file$            % transfer it to the server
    popup "New file created", 0, 0, 0
endif

! Now read the current contents of the online text chat, and save that
! text to the file "chat.txt" on the local Android device:

ftp.get file$, "chat.txt"

! Next append some text to the downloaded text file above, indicating that
! the user has entered the chat ("_Name_ has entered the room").  Opening
! a text file with the "a" option APPENDS to the existing text in the file
! (as opposed to erasing what's already there).  Using concatenation, the
! text written to the "chat.txt" file is the combined value of the current
! date and time, the user's name, some static text, and a carriage return.
! Notice the use of the underscore character to continue the concatenation
! onto a second line:

text.open a, chattext, "chat.txt"
time y$, m$, d$, h$, n$, s$
text.writeln chattext, m$ + "-" + d$ + ", " + h$ + ":" + m$ + ": " + ~
    name$ + " has entered the room."
text.close chattext

! Now the program uses a forever-repeating Do/While loop to continually
! wait for user input, and to do appropriate things with that input.  The
! following "flag$" variable is set to "continue".  This variable will be
! used later to determine if the user wants to end the program:

flag$ = "continue"

do

    ! Clear the screen:

    cls

    ! Display a greeting and some instructions.  Notice the use of "\n"
    ! to print a carriage return (new line), and the underscore symbol
    ! (_) to continue concatenation onto numerous lines:

    print "------------------------------------\n" + ~
        "You are logged in as: " + name$ + "\n" + ~
        "Type 'room' to switch chat rooms.\n" + ~
        "Type 'quit' to end your chat.\n" + ~
        "Enter blank text (just click OK)\n" + ~
        "to periodically update the display.\n" + ~
        "------------------------------------"
    ! Read the updated messages that are currently in the "chat.txt" text
    ! file on the FTP server, assign the variable word currentChat$ to
    ! that text, and print the contents:

    ftp.get file$, "chat.txt"
    grabfile currentChat$, "chat.txt"
    print "Here's the current chat text at: " + file$
    print currentChat$

    ! Pause a few seconds to allow the user to read the chat:

    pause 5000

    ! Get some text to be added to the chat room, entered by the user.
    ! Save the entered text in the string variable "enteredText$":

    input "You say:", enteredText$

    ! The If/Elseif/Else structure below is used to check for commands in
    ! the text entered by the user.  If the user enters "quit", empty
    ! text, or some other text to be added to the conversation,
    ! appropriate actions occur:

    if enteredText$ = "quit"

        ! If the user typed "quit", stop the forever loop (which will exit
        ! the program).  You could alternately use the "d_u.break"
        ! function here:

        flag$ = "quit"

    elseif enteredText$ <> ""

        ! If the user entered some text (anything other than blank text),
        ! Concatenate the text with the user's name and the static text
        ! " says: ".  Store that concatenated text in the string variable
        ! "sentMessage$":

        sentMessage$ = name$ + " says: " + enteredText$

        ! Now download the current chat text again (to make sure you're
        ! working with the most recent update(s)), append the message
        ! above to it, and then upload the appended text back to the FTP
        ! server:

        ftp.get file$, "chat.txt"
        text.open a, chattext, "chat.txt"     % remember, open to APPEND
        text.writeln chattext, sentMessage$
        text.close chattext
        ftp.put "chat.txt", file$

    endif

    ! The only other possible option at this point is that the user
    ! entered nothing (blank text, "").  In that case, the loop is simply
    ! repeated, so the current chat text is again downloaded, the display
    ! is updated, user input requested, the If/Then/Else conditions
    ! evaluated again, etc.  This goes on until the user types "quit".

until flag$ = "quit"
```

```
! When the forever repeating loop is exited, append a final message to the
! chat text, close the FTP connectin, and end the program:

cls
print "Goodbye!"
ftp.get file$, "chat.txt"
text.open a, chattext, "chat.txt"     % remember, open to APPEND
text.writeln chattext, h$ + ":" + m$ + ": " + name$ + ~
    " has left the room."
text.close chattext
ftp.put "chat.txt", file$
ftp.close
pause 1000
end
```

The forever repeating loop structure used in this program is a very common outline for apps that repeatedly accept input from users, process the incoming data, and display resulting output data. It's especially useful in games that continually move lists of graphic items around a screen (more on that later in this tutorial).

Here's the entire program, without comments:

```
input "FTP URL", site$, "ftp.site.com"
input "Folder", folder$, "./public_html/"
input "File", file$, "chat.txt"
input "User Name", username$, "user"
input "Password", password$, "pass"
input "Your Screen Name", name$, "Name"
ftp.open site$, 21, username$, password$
ftp.cd folder$
exists = 0
ftp.dir files
list.size files, s
for i = 1 to s
    list.get files, i, page$
    if file$ = page$ then exists = 1
next i
if exists = 0
    console.save "temp.txt"
    ftp.put "temp.txt", file$
    popup "New file created", 0, 0, 0
endif
ftp.get file$, "chat.txt"
text.open a, chattext, "chat.txt"
time y$, m$, d$, h$, n$, s$
text.writeln chattext, m$ + "-" + d$ + ", " + h$ + ":" + m$ + ": " + ~
    name$ + " has entered the room."
text.close chattext
flag$ = "continue"
do
    cls
    print "------------------------------------\n" + ~
        "You are logged in as: " + name$ + "\n" + ~
        "Type 'room' to switch chat rooms.\n" + ~
        "Type 'quit' to end your chat.\n" + ~
        "Enter blank text (just click OK)\n" + ~
        "to periodically update the display.\n" + ~
        "------------------------------------"
    ftp.get file$, "chat.txt"
    grabfile currentChat$, "chat.txt"
    print "Here's the current chat text at: " + file$
    print currentChat$
    pause 5000
    input "You say:", enteredText$
    if enteredText$ = "quit"
        flag$ = "quit"
    elseif enteredText$ <> ""
        sentMessage$ = name$ + " says: " + enteredText$
        ftp.get file$, "chat.txt"
        text.open a, chattext, "chat.txt"     % remember, open to APPEND
        text.writeln chattext, sentMessage$
        text.close chattext
        ftp.put "chat.txt", file$
    endif
until flag$ = "quit"
cls
print "Goodbye!"
ftp.get file$, "chat.txt"
text.open a, chattext, "chat.txt"     % remember, open to APPEND
text.writeln chattext, h$ + ":" + m$ + ": " + name$ + ~
    " has left the room."
text.close chattext
ftp.put "chat.txt", file$
ftp.close
pause 1000
end
```

9.5 Blogger

This program is another FTP example that allows users to create and add entries to an online blog page. The user enters a title, blog text, and a daily URL link of interest. The program determines the current date and time. All of those pieces of text are concatenated together, along with the necessary HTML layout code, and appended to a downloaded copy of the previous blog content. That code is then uploaded back to the user's web server to be viewed publicly. Be sure to edit the FTP and HTML URL variables so they represent actual user account information.

```
! Store the FTP account address, folder/filename, username and password
! in variables (for easy configuration):

input "FTP URL", site$, "ftpsite.com"
input "Folder", folder$, "./public_html/"
input "Filename", page$, "blog.html"
input "User Name", username$, "user"
input "Password", password$, "pass"
input "HTTP URL", url$, "http://site.com/blog.html"

! This portion of code checks to see if the page$ file already exists
! on the server:

ftp.open site$, 21, username$, password$
ftp.cd folder$
exists = 0
ftp.dir files
list.size files, s
for i = 1 to s
    list.get files, i, file$
    if file$ = page$ then exists = 1
next i

! If the file doesn't exist, create it:
```

```
if exists = 0
    console.save "temp.txt"
    ftp.put "temp.txt", page$
    popup "New file created", 0, 0, 0
endif

! Get a blog title, URL, and some blog text from the user.  Store that
! text in the variables blogtitle$ and blogtext$:

input "Blog Title:", blogtitle$, "Title"
input "URL:", blogurl$, "http://site.com"
text.input blogtext$, "Check out today's link!"

! Get the current date and time:

time y$, m$, d$, h$, n$, s$

! Concatenate the text entered above, and the date/time, with the HTML
! code required to display the blog entry.  Save the constructed HTML
! in the variable newblog$.  Notice the use of the underscore
! character to continue the string concatenation onto multiple lines:

newblog$ = "<h1>" + blogtitle$ + "</h1>" + "Blog entry created: " + ~
    m$ + "-" + d$ + "-" + y$ + " &nbsp; &nbsp; " + h$ + ":" + n$ + ~
    ":" + s$ + "<br><br><a href=\"" + blogurl$ + "\">" + blogurl$ + ~
    "</a><br><br>" + "<center><table width=80%><tr><td><pre><strong>" + ~
    blogtext$ + "</strong></pre></td></tr></table></center><br><hr>"

! Download the existing full HTML blog.  Save it in the file temp.txt:

ftp.get page$, "temp.txt"

! Load the downloaded blog text into the variable previousblog$:

grabfile previousblog$, "temp.txt"

! Write the concatenated new and old blog text to the file temp.txt:

text.open w, filename, "temp.txt"
text.writeln filename, newblog$ + previousblog$
text.close filename

! Upload the updated blog file to the web server:

ftp.put "temp.txt", page$
ftp.close

! Alert the user with a message, then open the blog in the browser:

popup "Saved", 0, 0, 1
browse url$
```

Here's the full program without comments:

```
input "FTP URL", site$, "ftpsite.com"
input "Folder", folder$, "./public_html/"
input "Filename", page$, "blog.html"
input "User Name", username$, "user"
input "Password", password$, "pass"
input "HTTP URL", url$, "http://site.com/blog.html"
ftp.open site$, 21, username$, password$
ftp.cd folder$
exists = 0
ftp.dir files
list.size files, s
for i = 1 to s
    list.get files, i, file$
    if file$ = page$ then exists = 1
next i
if exists = 0
    console.save "temp.txt"
    ftp.put "temp.txt", page$
    popup "New file created", 0, 0, 0
endif
input "Blog Title:", blogtitle$, "Title"
input "URL:", blogurl$, "http://site.com"
text.input blogtext$, "Check out today's link!"
time y$, m$, d$, h$, n$, s$
newblog$ = "<h1>" + blogtitle$ + "</h1>" + "Blog entry created: " + ~
    m$ + "-" + d$ + "-" + y$ + " &nbsp; &nbsp; " + h$ + ":" + n$ + ~
    ":" + s$ + "<br><br><a href=\"" + blogurl$ + "\">" + blogurl$ + ~
    "</a><br><br>" + "<center><table width=80%><tr><td><pre><strong>" + ~
    blogtext$ + "</strong></pre></td></tr></table></center><br><hr>"
ftp.get page$, "temp.txt"
grabfile previousblog$, "temp.txt"
text.open w, filename, "temp.txt"
text.writeln filename, newblog$ + previousblog$
text.close filename
ftp.put "temp.txt", page$
ftp.close
popup "Saved", 0, 0, 1
browse url$
```

9.6 Recipe Database

Here is a recipe database app that allows you to create, edit, search, store, retrieve, and view recipes. You can use this program as an outline for creating apps that store any sort of related information. You'll use the techniques of list management, file management, menu selection, text editing, serialization, subroutines, and others, often in your RFO Basic programs, so study this example carefully:

```
! First, check to see if the data file exists:

file.exists b, "recipes.txt"

! If the file doesn't exist, create it:

if b=0

! Create a new list structure, then add some default recipe titles,
! ingredients, and instructions.  Notice the use of the tilde character
! (~) to extend the list contents across multiple lines.  The current
! state of the list.add and array.load functions do not allow string
! concatenation using the tilde character within the function parameters.
! In order to add concatenated strings to a list, save the concatenated
! strings to a variable before executing the list.add function (see
! http://rfobasic.freeforums.org/post6133.html?hilit=#p6133 for a
! discussion of this topic):

    longstring$ = "Remove entree from " + ~
    "packaging\nHeat in microwave\nCool and eat with fork"
    list.create s, newrecipes
    list.add newrecipes, "Oatmeal", "Oatmeal\nWater\nMaple Syrup" ~
    "Boil water\nAdd oatmeal\nAdd maple syrup\nCool and eat" ~
    "Frozen Dinner", "Frozen Entree\nFork", longstring$

! Serialize the data into a single string, as shown earlier in the
! tutorial.  The serialized data is concatenated using the string
! separater "###", and saved in the variable savedata$:

    list.size newrecipes, size
    savedata$ = ""
    for i = 1 to size
        list.get newrecipes, i, m$
        if i = size
            savedata$ = savedata$ + m$
        else
            savedata$ = savedata$ + m$ + "###"
        endif
    next i

! Save the serialized data to the "recipes.txt" text file, then close the
! if conditional structure:

    text.open w, myfile, "recipes.txt"
    text.writeln myfile, savedata$
    text.close myfile
endif

! Load the saved data file (the new one, if just created above, or an old
! one, if previously saved by a user):

loadSavedRecipes:

    ! Read the serialized text string:

    grabfile serialdata$, "recipes.txt"

    ! Create a blank array:

    array.delete recipe$[]

    ! Split the string at the "###" characters:

    split recipe$[], serialdata$, "###"

    ! Convert the array into a list structure:

    list.create s, recipes
    list.add.array recipes, recipe$[]
! Create a separate list containing just the recipe titles (every
! third item in the list):
list.size recipes, size
list.create s, titles
for i = 1 to size step 3
  list.get recipes, i, n$
  list.add titles, n$
next i

! Display the title, ingredients and instructions for one recipe selected
! by the user:
```

```
viewRecipe:

! Add an option to the recipe titles list to create a new recipe
! (only if that option hasn't already been added):

list.search titles, "Create New Recipe...", c
if c = 0
    list.add titles, "Create New Recipe..."
endif

! Allow the user to select a recipe from the titles list:

select indx, titles, "Select a recipe to view/edit (hold to quit)", d

! If the user long-clicks, end the program:

if d = 1 then end

! If the user chooses to create a new recipe, jump to the appropriate
! subroutine:

list.get titles, indx, rcp$
if rcp$ = "Create New Recipe..."
    goto createNewRecipe
endif

! Otherwise, clear the screen and begin the recipe display routine:

cls

! Pay particular attention to this line.  Remember that every 3 items
! in the recipe list structure are: title/ingredients/instructions, so
! the index of the selected title is going to be the first of every
! 3 items.  If you pick the second title from the title list, it's
! position in the recipes list is 2 * 3 - 2 (second group of 3 items,
! 2 back from the index of that last item).  Mathematically, that can
! be represented like this:

s = indx * 3 - 2

! Pick out and print the selected title from the recipe list (found at
! the index above):

list.get recipes, s, n$
print "TITLE:"
print n$ + "\n"

! Pick out and print the selected ingrediants from the recipe list
! (found at the index above +1):

list.get recipes, s+1, a$
print "INGREDIENTS:"
print a$ + "\n"

! Pick out and print the selected ingrediants from the recipe list
! (found at the index above +2):

list.get recipes, s+2, p$
print "COOKING INSTRUCTIONS:"
print p$
print ""
```

```
! Give the use the option to click a key to continue (make sure the
! onscreen keyboard is showing):

print "(Click any key to continue...)"
kb.hide
pause 1000
kb.toggle
pause 1000
let done = 0
do
    inkey$ k$
    if k$ <> "@" then let done = 1
until done

! When the viewRecipe routine is done, go back and do it all again:

goto viewRecipe

! Here's the subroutine to create a new recipe.  It's basically the same
! as the code used earlier to create a new data file.  It simply requests
! title, ingredients, and instructions texts from the user.  Those texts
! are assigned variable labels and then added to the existing recipe list
! structure.  The recipe list is then serialized into a long concatenated
! string and re-saved to the file "recipes.txt".  When this routine is
! complete, the program returns back to the loadSavedRecipes routine to
! begin again:

createNewRecipe:
input "Title:", title$, ""
text.input ingredients$, "(Type ingredients here)"
text.input instructions$, "(Type instructions here)"
list.add recipes, title$
list.add recipes, ingredients$
list.add recipes, instructions$
list.size recipes, size
savedata$ = ""
for i = 1 to size
    list.get recipes, i, m$
    if i = size
        savedata$ = savedata$ + m$
    else
        savedata$ = savedata$ + m$ + "###"
    endif
next i
text.open w, myfile, "recipes.txt"
text.writeln myfile, savedata$
text.close myfile
goto loadSavedRecipes
```

Here's the code without comments:

```
file.exists b, "recipes.txt"
if b=0
    longstring$ = "Remove entree from " + ~
    "packaging\nHeat in microwave\nCool and eat with fork"
    list.create s, newrecipes
    list.add newrecipes, "Oatmeal", "Oatmeal\nWater\nMaple Syrup" ~
    "Boil water\nAdd oatmeal\nAdd maple syrup\nCool and eat" ~
    "Frozen Dinner", "Frozen Entree\nFork", longstring$
    list.size newrecipes, size
    savedata$ = ""
    for i = 1 to size
        list.get newrecipes, i, m$
        if i = size
            savedata$ = savedata$ + m$
        else
            savedata$ = savedata$ + m$ + "###"
        endif
    next i
    text.open w, myfile, "recipes.txt"
    text.writeln myfile, savedata$
    text.close myfile
endif
loadSavedRecipes:
    grabfile serialdata$, "recipes.txt"
    array.delete recipe$[]
    split recipe$[], serialdata$, "###"
    list.create s, recipes
    list.add.array recipes, recipe$[]
    list.size recipes, size
    list.create s, titles
    for i = 1 to size step 3
      list.get recipes, i, n$
      list.add titles, n$
    next i
viewRecipe:
    list.search titles, "Create New Recipe...", c
    if c = 0
        list.add titles, "Create New Recipe..."
    endif
    select indx, titles, "Select a recipe to view/edit", d
    if d = 1 then end
    list.get titles, indx, rcp$
    if rcp$ = "Create New Recipe..."
        goto createNewRecipe
    endif
    cls
    s = indx * 3 - 2
    list.get recipes, s, n$
    print "TITLE:"
    print n$ + "\n"
    list.get recipes, s+1, a$
    print "INGREDIENTS:"
    print a$ + "\n"
    list.get recipes, s+2, p$
    print "COOKING INSTRUCTIONS:"
    print p$
    print ""
    print "(Click any key to continue...)"
    kb.hide
    pause 1000
    kb.toggle
    pause 1000
    let done = 0
    do
        inkey$ k$
        if k$ <> "@" then let done = 1
    until done
goto viewRecipe
createNewRecipe:
    input "Title:", title$, ""
    text.input ingredients$, "(Type ingredients here)"
```

```
        text.input instructions$, "(Type instructions here)"
        list.add recipes, title$
        list.add recipes, ingredients$
        list.add recipes, instructions$
        list.size recipes, size
        savedata$ = ""
        for i = 1 to size
            list.get recipes, i, m$
            if i = size
                savedata$ = savedata$ + m$
            else
                savedata$ = savedata$ + m$ + "###"
            endif
        next i
        text.open w, myfile, "recipes.txt"
        text.writeln myfile, savedata$
        text.close myfile
    goto loadSavedRecipes
```

This may seem like a lot of code for such a simple program, but the above app demonstrates some absolutely essential code patterns. Learning to think in terms of variables, data list structures, loops, and conditional structures is fundamental to achieving the overwhelming majority of useful programming goals. Saving serialized data to a file is useful if you need to share with other devices, upload the data to a web server to be parsed and entered into a database, display it online, etc. In a more straightforward way, this app could be easily converted to store contact information, retail inventory information, or any other sort of personal or business information, basically by renaming the field labels. For example, instead of "Title", "Ingredients", and "Cooking Instructions", the fields could be labeled "Item", "Size", and "Description", or "Name", "Phone Number(s), and "Address", etc. Try adjusting the loops, index variables, and list structures a bit to create an app that stores more than 3 fields. Next, we'll create a GUI version of this program.

10. HTML/Javascript GUIs

You've already seen how the RFO Basic "input" and "text.input" functions can be used to acquire text typed by users, the "select" function can be used to display and accept menu choices, and "print" can be used to output data. For the simple types of utility scripts that RFO Basic is perfectly suited to create, these input/output functions will often be all that are ever needed. To create more complex data entry screens, or to build more visually appealing Graphic User Interfaces ("GUI"s), RFO Basic allows you to use standard HTML and Javascript pages to display buttons, text fields, drop down selection lists, check boxes, text areas, images, and other familiar form elements. You can use any of the common Javascript libraries supported by the Android browser (jQuery Mobile, Sencha Touch, jQTouch), to provide rich graphic user interface interactions for your RFO Basic apps.

10.1 An HTML Crash Course

Covering HTML and Javascript in depth is beyond the scope of this tutorial, but a quick crash course for absolute beginners will provide enough understanding to be very useful. To learn more than basic HTML, search Google for "HTML tutorial", "javascript tutorial", and for topics such as "mobile ui framework" - there are thousands of resources for developers with every level of experience.

HTML is the layout language used to format text and GUI elements on all web pages. HTML is not a programming language - it cannot evaluate conditional expressions, it has no facility to control phone hardware such as sensors, camera, etc., it *can't process or manipulate data* in any useful way, etc. 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. RFO Basic allows you to manipulate data and control the phone hardware, so the two are a perfect compliment to one another.

10.1.1 Tags

In HTML, items on a web page are enclosed between start and end "tags":

```
<START TAG>Some text or graphic element on a web page</END 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*.

Notice that every

```
<opening tag>
```

in HTML code is followed by a corresponding

```
</closing tag>
```

The closing tag starts with the "/" character. 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.

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>
```

The above code produces the following result:

First Row
Second Row
Third Row

Notice that the entire table above is enclosed in opening and closing < T A B L E > tags, and each row is enclosed in opening/closing < T R > (table row) and < T D > (table data) tags.

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 named "yourpage.html" in /sdcard/rfo-basic/data/ on your Android devive, and surf to file:///sdcard/rfo-basic/data/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. This document is simply saved on a server which is publicly available at the http://rfobasic.com URL.

10.2 HTML Forms - Basic GUI Widgets

The following HTML example contains a "form" tag inside the standard HTML head and body layout. Inside the form tags are 2 text input (field) tags, a multiline textarea tag, a dropdown selection tag allowing the user to select one day from the 7 weekdays, 4 check box tags allowing the user to select any number of fruits, and a submit button tag. The "
" tags are required to produce carriage returns (new lines). Without them, every item would flow from left to right across the screen, and then wrap around to fill the browser page:

```
<html>
    <head><title>Data Entry Form</title></head>
    <body>
        <form action="form">
            Name: <br>
            <input type="text" name="name"><br><br>
            Email: <br>
            <input type="text" name="email"><br><br>
            Message: <br>
            <textarea cols=25 name="message" rows=5>Hi.</textarea><br><br>
            Preferred Day:<br>
            <select name="day">
                <option>Monday<option>Tuesday<option>Wednesday
                <option>Thursday<option>Friday<option>Saturday
                <option>Sunday</option>
            </select><br><br>
            Favorite Fruit(s):<br>
            <input type="checkbox" name="f1" value="fig" checked>fig<br>
            <input type="checkbox" name="f2" value="apple">apple<br>
            <input type="checkbox" name="f3" value="orange">orange<br>
            <input type="checkbox" name="f4" value="banana">banana<br><br>
            <input type="submit" name="submit" value="submit">
        </form>
    </body>
</html>
```

Copy the code above into a blank text document, save it as "myform.html", and then open that document with your browser. You'll see the following:

Name:

Email:

Message:

```
Hi.
```

Preferred Day:

Favorite Fruit(s):

- ✓ fig
- ☐ apple
- ☐ orange
- ☐ banana

The form elements above are enough to satisfy the overwhelming majority of data input requirements for simple programs. For more information about HTML form tags, see http://www.w3schools.com/html/html_forms.asp.

10.3 Using HTML Form Data

You can use the data entered into any form as follows (to use this example, save the HTML form above as */sdcard/rfo-basic/data/myform.html*):

```
html.open
html.load.url "file:///sdcard/rfo-basic/data/myform.html"
do
    html.get.datalink data$
until data$ <> ""
data$ = mid$(data$,5)
print "Form data: "+data$
```

All HTML forms use the "&" character to serialize submitted data into one long concatenated string. You can use the "split" function to create an array containing each element of the submitted data, by separating at each "&" character:

```
split submitted$[], data$, "&"
select x, submitted$[], ""
```

Here's a full a example that cleans up the returned values a bit more, removing the URL of the form and the "?" character, splitting at the "&" character, and replacing any "+" (space) and "%0D%0A" (newline) characters that are included in the serialized data string:

```
html.open
html.load.url "file:///sdcard/rfo-basic/data/myform.html"
do
    html.get.datalink data$
until data$ <> ""

! Get the index of where the "?" character is found in the data string:

indx = is_in("?", data$)

! Cut off all the characters up to and including the "?" character:
```

```
data$ = mid$(data$, (indx + 1))

! Split at the & character:

split submitted$[], data$, "&"

! Allow the user to choose from the split data chunks:

select chosen, submitted$[], "Choose a field:"
chosen$ = submitted$[chosen]

! Split the data chunks at the "=" character:

split chosen$[], chosen$, "="

! The first item of the two split above is the field label:

field$ = chosen$[1]

! The second is the value:

value$ = chosen$[2]

! Space and return characters in the value text must be decoded:

value$ = replace$(value$, "+", " ")
value$ = replace$(value$, "%0D%0A", "\n")

! Print the field and the value:

print "field = " + field$
print "value = " + value$
```

You can use the replace$ function to remove *any other HTML codes* that might be potentially entered by a user into GUI HTML forms.

10.4 Dynamically Creating GUI Layouts

The "html.load.url" function introduced above can read an HTML file from the file system, *or from an online web server*:

```
html.load.url "http://site.com/myform.html"
```

If you need to create a GUI with a dynamically changing layout based upon user input or any other variable condition, use the "html.load.string" function. This allows you to concatenate an HTML string, and then display it as a GUI:

```
input "Name:", name$

frm$ = "<form action=\"form\">Name: <br>" + ~
    "<input type=\"text\" name=\"name\" value=\"" + ~
    name$ + "\"><br><br>" + ~
    "<input type=\"submit\" name=\"submit\" value=\"submit\">" + ~
    "</form>"
html.open
html.load.string frm$
do
```

```
    html.get.datalink data$
until data$ <> ""
data$ = mid$(data$,5)
print "Form data: "+data$
```

The technique above provides a versatile way to create GUIs programmatically during runtime. It also provides a way to store code for GUI layouts directly within program code, so that no additional HTML files need to be distributed separately from the script. Notice the use of the "+ ~" characters to concatenate the multiline string, and the use of the "\" character to include quotes within the quoted text.

Another useful technique is to read an exising HTML file, and use the "replace$" function to dynamically alter the default layout. For example, the following code reads the myform.html document and changes the page color:

```
grabfile frm$, "myform.html"
! Replace the body tag with a body tag that contains a background color:
frm$ = replace$(frm$, "<body>", "<body bgcolor=\"#E6E6FA\">")
html.open
html.load.string frm$
do
    html.get.datalink data$
until data$ <> ""
data$ = mid$(data$,5)
print "Form data: " + data$
```

The following code does the exact same thing, but loads the HTML from a concatenated string, instead of from the external file:

```
frm$ = "<head><title>Data Entry Form</title></head>" + ~
"<body>" + ~
    "<form action=\"form\">" + ~
        "Name: <br>" + ~
        "<input type=\"text\" name=\"name\"><br><br>" + ~
        "Email: <br>" + ~
        "<input type=\"text\" name=\"email\"><br><br>" + ~
        "Message: <br>" + ~
        "<textarea cols=25 name=\"message\" rows=5>Hi." + ~
        "    </textarea><br><br>" + ~
        "Preferred Day:<br>" + ~
        "<select name=\"day\">" + ~
            "<option>Monday<option>Tuesday<option>Wednesday" + ~
            "<option>Thursday<option>Friday<option>Saturday" + ~
            "<option>Sunday</option>" + ~
        "</select><br><br>" + ~
        "Favorite Fruit(s):<br>" + ~
        "<input type=\"checkbox\" name=\"f1\"" + ~
        "    value=\"fig\" checked>fig<br>" + ~
        "<input type=\"checkbox\" name=\"f2\"" + ~
        "    value=\"apple\">apple<br>" + ~
        "<input type=\"checkbox\" name=\"f3\"" + ~
        "    value=\"orange\">orange<br>" + ~
        "<input type=\"checkbox\" name=\"f4\"" + ~
        "    value=\"banana\">banana<br><br>" + ~
        "<input type=\"submit\" name=\"submit\" value=\"submit\">" + ~
    "</form>" + ~
"</body>"
frm$ = replace$(frm$, "<body>", "<body bgcolor=\"#E6E6FA\">")
html.open
html.load.string frm$
do
    html.get.datalink data$
until data$ <> ""
data$ = mid$(data$,5)
print "Form data: " + data$
```

Using replace$ in this way, it's possible to change *any* predefined HTML in an existing GUI layout, add default data to forms, or make any other alterations, all at runtime, based on data entered by the user or any other dynamic factor(s). This is very powerful!

Learning more HTML, CSS, and Javascript is helpful if you want to create impressively complex and/or responsive layouts, but the form elements you've seen so far provide all the essential components needed to collect data from users, in a way that's familiar and effective.

10.5 Some More Essential HTML Tags

Here are a few more examples demonstrating useful HTML tags. Paste each example into a separate code file, save the files with a ".html" extension, and then open each file in your web browser to see how the tags affect layout and formatting. The following examples do not improve your ability to collect user data. They simply demonstrate adjustments to properties such as font size, item positioning, added graphics, links, etc.:

```
<h1>This is some header text</h1>
<h3>This header text is smaller</h3>
In HTML, all spaces, tabs, newlines and other white spaces
are compressed (i.e., this text is all printed on the same
line, and wrapped to the length of the screen, and these
spaces             are all compressed to one space.  If
you want to explicitly display a certain number of spaces,
use these characters:  &nbsp; &nbsp;  For newlines (carriage
returns), use the "br" tag: <br><br>
To adjust text size and color, use the "font" tag: <br><br>
<font size=1 color=blue>Little blue text</font><br>
<font size=6 color=red>Big red text</font><br><br>
To display a link to a web site, use the "a" tag: <br><br>
<a href="http://yahoo.com">yahoo.com</a><br><br>
To display images, use the "img" tag: <br><br>
<img src="http://laughton.com/basic/logo.png"><br><br>
To make an image link to a URL, use "img" and "a": <br><br>
<a href="http://laughton.com/basic/" target=_blank>
    click the image:<br>
    <img src="http://laughton.com/basic/logo.png">
</a><br><br>
<strong>This text is bolded.</strong><br><br>
<pre>
    The "pre" tag displays preformatted text.
    Spaces, tabs, newlines, etc. are kept intact.
</pre>
You can add a horizontal separator line with the "hr" tag:
<br><br><hr width=80%><br>
Center anything on a page with the "center" tag: <br><br>
<center>Centered Text</center><br>
Tables are used to display rows and columns of data.  They
can also be used to align portions of an HTML page in certain
areas of the screen (i.e., to layout menu bar areas, headers,
main content areas, and footers): <br><br>
<center>
    <table border=0 cellpadding=10 width=80%>
        <tr>
```

```
            <td width=33%>"tr" tags</td>
            <td width=33%>are</td>
            <td width=33%>rows</td>
        </tr>
        <tr>
            <td width=33%>"td" tags</td>
            <td width=33%>are</td>
            <td width=33%>columns</td>
        </tr>
        <tr>
            <td valign=top colspan=2 bgcolor=#CCCCCC>
                You can set "td"s to span as many columns as
                needed, using the "colspan" setting, and you
                can use the "valign" property to set where
                in the cell text is placed.  "bgcolor" sets
                the cell's background color.
            </td>
            <td width=20 bgcolor=#000000>
                <font color=white size=2>
                    You can set size properties of all different
                    parts of the table, using either percentage
                    or pixel values.
                </font>
            </td>
        </tr>
    </table>
</center>
Try resizing the page to see how centering, text wrapping and
table layouts work.
```

The following two examples demonstrate how to use tables to layout items on a page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Generic HTML Page With Tables</TITLE>
  <META
        http-equiv=Content-Type content="text/html;
        charset=windows-1252"
  >
</HEAD>
<BODY bgColor=#CCCCCC>
    <TABLE align=center background="" border=0
        cellPadding=20 cellSpacing=2 height="95%" width="85%"
    >
        <TR>
            <TD background="" bgColor=white vAlign=top>
                Your page content goes here.
            </TD>
        </TR>
    </TABLE>
    <TABLE align=center background="" border=0 cellPadding=2
        cellSpacing=2 width="85%" height="5%"
    >
        <TR>
            <TD background="" cellPadding=2 bgColor=#000000 height=5>
                <P align=center>
                    <FONT color=white size=1>
                        Copyright Š 2009 Yoursite.com.
                        All rights reserved.
                    </FONT>
                </P>
            </TD>
        </TR>
    </TABLE>
</BODY>
</HTML>


<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>Table Based Page Layout,  No Menu</TITLE>
    <META http-equiv=Content-Type content="text/html;
         charset=windows-1252"
    >
    <META
        HTTP-EQUIV="Page-Enter"
        CONTENT="RevealTrans(Duration=2,Transition=14)"
    >
    <STYLE TYPE="text/css">
        a:hover {
            color: blue;
            text-decoration: underline;
            background: #F9EDED
        }
    </STYLE>
</HEAD>
<BODY bgColor=#000000>
<TABLE align=center background="" border=0 cellPadding=2 cellSpacing=2
    height="100%" width="95%"
>
    <TR>
        <TD background="" bgColor=lightgrey vAlign=top>
            <TABLE background="" border=0 cellPadding=2 cellSpacing=2
                height="100%" width="100%"
            >
                <TR>
                    <TD background="" bgColor=black height=10>
                        <P align=center>
                            <a href="./index.html">
                                <IMG align=baseline alt="HEADER IMAGE"
                                    border=0 hspace=0 src="header.jpg"
                                >
                            </a>
                        </P>
                    </TD>
                </TR>
                <TR>
                <TD background="" vAlign=top>
                    <TABLE bgColor=#ffffff border=0 cellPadding=2
                        cellSpacing=2 height="100%" width="100%"
                    >
                        <TR>
                            <TD bgColor=lightgrey height=20>
                                <center></center>
                            </TD>
                            <TD bgColor=lightgrey height=10 width="90%">
                                <center>
                                    <a href="./Home.html">
                                        Home
                                    </a> : Contact
                                </center>
                            </TD>
                            <TD bgColor=lightgrey height=20>
                                <center></center>
                            </td>
                        </TR>
                        <TR>
                            <TD background="" colSpan=3 height="100%">
                                <TABLE border=0 cellPadding=15
                                    cellSpacing=2 height="100%"
                                    width="100%"
                                >
                                    <TR>
                                        <TD background="" vAlign=top>
                                            YOUR PAGE CONTENT GOES HERE
                                        </TD>
                                    </TR>
                                </TABLE>
                            </TD>
                        </TR>
                    </TABLE>
                </TD>
            </TR>
            <TR>
                <TD background="" bgColor=#000000 height=5>
                    <P align=center>
                        <FONT color=white size=1>
                            Copyright Š 2010 This Web Site.
                            All rights reserved.
                        </FONT>
                    </P>
                </TD>
            </TR>
        </TABLE>
    </TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

10.6 Responding to Links, Back Buttons, and Errors

The following example is taken from the De Re Basic manual by Paul Laughton. It shows how to respond to all the possible HTML/Javascript UI options:

```
html.open
html.load.url "file:///sdcard/rfo-basic/data/htmlDemo1.html"
xnextUserAction:
do
    html.get.datalink data$
until data$ <> ""
type$ = left$(data$, 4)
data$ = mid$(data$,5)
sw.begin type$
    sw.case "BAK:"
        print "BACK key: " + data$
        if data$ = "1" then html.go.back else end
    sw.break
    sw.case "LNK:"
        print "Hyperlink selected: "+ data$
        html.load.url data$
    sw.break
    sw.case "ERR:"
        print "Error: " + data$
    sw.break
    sw.case "DAT:"
        print "User data: " + data$
        if  left$(data$, 4) = "Exit"
            print "User ended demo."
            html.close
            end
        endif
        if data$ = "STT"
            stt.results thelist
            list.get thelist, 1, thetext$
            thetext$ = "You said: " + thetext$
            insert$ = "javascript:text(\""+thetext$+"\")"
            print insert$
            html.load.url insert$
        endif
    sw.break
    sw.case "FOR:"
        print "Form data: "+data$
        end
    sw.break
    sw.case "DNL:"
        print "Download: " + data$
        array.delete p$[]
        split p$[], data$, "/"
        array.length l, p$[]
        fn$ = p$[l]
        html.load.string "<html> Starting download of " + fn$ + "</html>"
        byte.open r,f,data$
        pause 2000
        html.go.back
        byte.copy f,fn$
        byte.close
    sw.break
    sw.default
        print "Unexpected data type:", data$
        end
sw.end
```

10.7 Recipe Database #2 - A Little GUI Data Storage and Retrievel App

The program below is a variation of the recipe script provided in the previous section. To improve user input and data display functionality, a long string of HTML is concatenated to form the GUI interface. The replace$ function is used to dynamically place data into the text input and textarea fields, as needed during runtime. Notice that the only changed code from the original program are lines that contain print, input, and text.input functions:

```
! Here's the default GUI layout.  It contains the characters "tttt",
! "iiii", and "cccc", which will be replaced with live title, ingredient,
! and instruction data as needed:

gui$ = "<form action=\"form\">" + ~
    "Title: <br>" + ~
    "<input type=\"text\" name=\"title\" value=\"tttt\"><br><br>" + ~
    "Ingredients: <br>" + ~
    "<textarea cols=25 name=\"ingred\" rows=5>iiii</textarea><br><br>" + ~
    "Instructions: <br>" + ~
    "<textarea cols=25 name=\"instru\" rows=5>cccc</textarea><br><br>" + ~
    "<input type=\"submit\" name=\"submit\" value=\"Done\">" + ~
"</form>"

! This function is used to replace formatted characters such as spaces and
! newlines in the submitted form text:

fn.def pickvalue$(submit$)
    split submit$[], submit$, "="
    value$ = submit$[2]
    value$ = replace$(value$, "+", " ")
    value$ = replace$(value$, "%0D%0A", "\n")
    value$ = replace$(value$, "%27", "'")
    fn.rtn value$
fn.end

! The main program starts here:

file.exists b, "recipes.txt"
if b=0
    longstring$ = "Remove entree from " + ~
    "packaging\nHeat in microwave\nCool and eat with fork"
    list.create s, newrecipes
    list.add newrecipes, "Oatmeal", "Oatmeal\nWater\nMaple Syrup" ~
    "Boil water\nAdd oatmeal\nAdd maple syrup\nCool and eat" ~
    "Frozen Dinner", "Frozen Entree\nFork", longstring$
    list.size newrecipes, size
    savedata$ = ""
    for i = 1 to size
        list.get newrecipes, i, m$
        if i = size
            savedata$ = savedata$ + m$
        else
            savedata$ = savedata$ + m$ + "###"
        endif
    next i
    text.open w, myfile, "recipes.txt"
    text.writeln myfile, savedata$
    text.close myfile
endif
loadSavedRecipes:
    grabfile serialdata$, "recipes.txt"
    array.delete recipe$[]
    split recipe$[], serialdata$, "###"
    list.create s, recipes
    list.add.array recipes, recipe$[]
    list.size recipes, size
    list.create s, titles
    for i = 1 to size step 3
      list.get recipes, i, n$
      list.add titles, n$
    next i
viewRecipe:
    list.search titles, "Create New Recipe...", c
    if c = 0
        list.add titles, "Create New Recipe..."
    endif
    select indx, titles, "Select a recipe to view/edit", d
    if d = 1 then end
    list.get titles, indx, rcp$
    if rcp$ = "Create New Recipe..."
```

```
        goto createNewRecipe
    endif
    cls
    s = indx * 3 - 2
    list.get recipes, s, t$
    list.get recipes, s+1, i$
    list.get recipes, s+2, c$
    gui2$ = gui$
    gui2$ = replace$(gui2$, "tttt", t$)
    gui2$ = replace$(gui2$, "iiii", i$)
    gui2$ = replace$(gui2$, "cccc", c$)
    html.open
    html.load.string gui2$
    do
        html.get.datalink data$
    until data$ <> ""
goto viewRecipe
createNewRecipe:
    gui2$ = gui$
    gui2$ = replace$(gui2$, "tttt", "")
    gui2$ = replace$(gui2$, "iiii", "")
    gui2$ = replace$(gui2$, "cccc", "")
    html.open
    html.load.string gui2$
    do
        html.get.datalink data$
    until data$ <> ""
    indx = is_in("?", data$)
    data$ = mid$(data$, (indx + 1))
    array.delete submitted$[]
    split submitted$[], data$, "&"
    title$ = pickvalue$(submitted$[1])
    ingredients$ = pickvalue$(submitted$[2])
    instructions$ = pickvalue$(submitted$[3])
    list.add recipes, title$
    list.add recipes, ingredients$
    list.add recipes, instructions$
    list.size recipes, size
    savedata$ = ""
    for i = 1 to size
        list.get recipes, i, m$
        if i = size
            savedata$ = savedata$ + m$
        else
            savedata$ = savedata$ + m$ + "###"
        endif
    next i
    text.open w, myfile, "recipes.txt"
    text.writeln myfile, savedata$
    text.close myfile
goto loadSavedRecipes
```

Here's the same program again with some useful parts of the code broken out into functions and subroutines. Isolating repeated sections of useful code such as this can help make your code shorter, more readable, and more reusable, as your programs become increasingly complex. You can use variations of the pickvalue$(), displayGUI$() and serialize$() functions in any program that makes use of HTML GUI forms:

```
gui$ = "<form action=\"form\">" + ~
    "Title: <br>" + ~
    "<input type=\"text\" name=\"title\" value=\"tttt\"><br><br>" + ~
    "Ingredients: <br>" + ~
    "<textarea cols=25 name=\"ingred\" rows=5>iiii</textarea><br><br>" + ~
    "Instructions: <br>" + ~
    "<textarea cols=25 name=\"instru\" rows=5>cccc</textarea><br><br>" + ~
    "<input type=\"submit\" name=\"submit\" value=\"Done\">" + ~
"</form>"
fn.def pickvalue$(submit$)
    split submit$[], submit$, "="
    value$ = submit$[2]
    value$ = replace$(value$, "+", " ")
    value$ = replace$(value$, "%0D%0A", "\n")
    value$ = replace$(value$, "%27", "'")
    fn.rtn value$
fn.end
fn.def displayGUI$(tt$, ii$, cc$, g$)
    gui2$ = g$
    gui2$ = replace$(gui2$, "tttt", tt$)
    gui2$ = replace$(gui2$, "iiii", ii$)
    gui2$ = replace$(gui2$, "cccc", cc$)
    html.open
    html.load.string gui2$
    do
        html.get.datalink data$
    until data$ <> ""
    fn.rtn data$
fn.end
fn.def serialize$(recipelist)
    list.size recipelist, size
    savedata$ = ""
    for i = 1 to size
        list.get recipelist, i, m$
        if i = size
            savedata$ = savedata$ + m$
        else
            savedata$ = savedata$ + m$ + "###"
        endif
    next i
    text.open w, myfile, "recipes.txt"
    text.writeln myfile, savedata$
    text.close myfile
fn.end
file.exists b, "recipes.txt"
if b=0
    longstring$ = "Remove entree from " + ~
    "packaging\nHeat in microwave\nCool and eat with fork"
    list.create s, newrecipes
    list.add newrecipes, "Oatmeal", "Oatmeal\nWater\nMaple Syrup" ~
    "Boil water\nAdd oatmeal\nAdd maple syrup\nCool and eat" ~
    "Frozen Dinner", "Frozen Entree\nFork", longstring$
    sss$ = serialize$(newrecipes)
endif
loadSavedRecipes:
    grabfile serialdata$, "recipes.txt"
    array.delete recipe$[]
    split recipe$[], serialdata$, "###"
    list.create s, recipes
    list.add.array recipes, recipe$[]
    list.size recipes, size
    list.create s, titles
    for i = 1 to size step 3
      list.get recipes, i, n$
```

```
        list.add titles, n$
    next i
viewRecipe:
    list.search titles, "Create New Recipe...", c
    if c = 0 then list.add titles, "Create New Recipe..."
    select indx, titles, "Select a recipe to view/edit", d
    if d = 1 then end
    list.get titles, indx, rcp$
    if rcp$ = "Create New Recipe..." then goto createNewRecipe
    cls
    s = indx * 3 - 2
    list.get recipes, s, t$
    list.get recipes, s+1, i$
    list.get recipes, s+2, c$
    data$ = displayGUI$(t$, i$, c$, gui$)
goto viewRecipe
createNewRecipe:
    data$ = displayGUI$("", "", "", gui$)
    indx = is_in("?", data$)
    data$ = mid$(data$, (indx + 1))
    array.delete submitted$[]
    split submitted$[], data$, "&"
    title$ = pickvalue$(submitted$[1])
    ingredients$ = pickvalue$(submitted$[2])
    instructions$ = pickvalue$(submitted$[3])
    list.add recipes, title$
    list.add recipes, ingredients$
    list.add recipes, instructions$
    sss$ = serialize$(recipes)
goto loadSavedRecipes
```

11. Graphics

11.1 Screen Setup

A graphic screen is created in RFO Basic using the "gr.open alpha, red, green, blue {, ShowStatusBar}" function (the alpha parameter is the transparency level, the other colors form the background color of the graphic screen):

```
gr.open 255, 255, 255, 255                % solid white background
```

To set the color and fill pattern used to draw graphics, use the "gr.color alpha, red, green, blue, style" function (for style, fill pattern 1=stroke(outline), 2=fill, 3=strokeandfill):

```
gr.color 255, 0, 0, 255, 0                % 0 transparency blue outline
```

11.2 Drawing Shapes

A number of functions allow you to place various graphic shapes on the screen:

```
gr.circle Object_number, x, y, radius
gr.line Object_number, x1, y1, x2, y2
gr.oval Object_number, left, top, right, bottom
gr.poly Object_number, List_pointer {,x,y}
gr.rect Object_number, left, top, right, bottom
gr.arc  Object_number, left, top, right, bottom, start_angle,
    sweep_angle, fill_mode
```

To place a 10x15 square at 50 pixels over and 40 pixels down from the left corner of the screen, labeled "rct1":

```
gr.rect rct1, 50, 40, 60, 15    % left, top, right, and bottom coordinates
```

11.3 Modifying Position and other Properties

To change any property (location, size, etc.) of any graphic object, use the gr.modify function. The following code moves the rectangle above 10 pixels to the right (left coordinate from position 50 to 60, right coordinate from 60 to 70):

```
gr.modify rct1, "left", 60
gr.modify rct1, "right", 70
```

11.4 Rendering Changes

Changes made to any graphic *do not appear on screen until you call the **gr.render** function*. In order to see any changes made to a graphic object, you must do this every time:

```
gr.render
```

11.5 Looping Graphic Modifications

You can use loops to move objects, using variables for the position values:

```
for i = 50 to 100 step 1          % move 50 pixels every time through loop
    gr.modify rct1, "left", i
    gr.modify rct1, "right", (i + 10)
    gr.render
    pause 1                              % adjust the animation speed here
next
```

11.6 Closing the Graphic Screen

Close the graphic screen using the gr.close function:

```
gr.close
```

11.7 An Animated Rectangle

Here's a complete program that puts together everything above to produce an animated rectangle which moves across the screen 50 pixels:

```
gr.open 255, 255, 255, 255
gr.color 255, 0, 0, 255, 0
gr.rect rct1, 50, 40, 60, 15          % remember: left, top, right, bottom
gr.render
for i = 50 to 100 step 1
    gr.modify rct1, "left", i
    gr.modify rct1, "right", (i + 10)
    gr.render
    pause 1
next
pause 1000
gr.close
```

11.8 Setting Orientation

You can set the orientation of the graphic screen using the gr.orientation function (default=landscape, 0=landscape, 1=portrait, -1=determined by sensor):

```
gr.orientation 1
```

11.9 Text on the Graphics Screen

You can place text on the graphic screen using the gr.text.draw function:

```
gr.text.draw txt1, 10, 10, "Hello Graphic World!"
```

Here is a program that moves some text across the screen, and changes the text to announce it's current position:

```
gr.open 255, 255, 255, 255
gr.color 255, 0, 0, 255, 0
gr.orientation 1
gr.text.draw txt1, 1, 60, "My current 'x' position is 1"
for i = 1 to 100 step 1
    gr.modify txt1, "x", i
    gr.modify txt1, "text", "My current 'x' position is " + str$(i)
    gr.render
    pause 1
next
pause 1000
gr.close
```

11.10 Loading Images

Create image spaces using the gr.bitmap.create functions. Load images from a file on your device using the gr.bitmap.load function. Display loaded images using the gr.bitmap.draw function:

```
gr.bitmap.create img1, 50, 50               % width and height parameters
gr.bitmap.load img1, "cartman.jpg"          % installed with RFO Basic
gr.bitmap.draw img1, img, 20, 30            % position parameters
```

Here is a program that moves the cartman.jpg image across the screen:

```
gr.open 255, 255, 255, 255
gr.color 255, 0, 0, 255, 0
gr.orientation 1
gr.bitmap.create img1, 50, 50
gr.bitmap.load img, "cartman.jpg"
gr.bitmap.draw img1, img, 20, 30
gr.render
for i = 20 to 200 step 5
    gr.modify img1, "x", I
    gr.render
    pause 1
next
pause 1000
gr.close
```

11.11 Switching Between Graphics and Text Console Screens

You can switch back and forth between the graphics screen and the normal RFO Basic text output console, using the gr.front function:

```
gr.front 0      % shows the text console
print "Get Ready for Stage 2..."
pause 2000
gr.front 1      % shows the graphics screen
gr.render
```

11.12 Touch

Use the gr.touch function to determine where a user has touched the screen. Use a do/until loop to wait for a touch:

```
do
    gr.touch touched, x, y
until touched
gr.front 0
print "You touched the screen at coordinate "; x; "x"; y
end
```

11.13 A Simple Sliding Puzzle Game

The following program is the classic 8-puzzle, in which you have 8 moveable tiles and 1 empty space. Touch the colored tile that you want to move into the empty space. This code uses many of the graphics features, conditional evaluations and flow control structures you've seen so far:

```
! Set up the screen:

gr.open 255, 255, 255, 255     % white background
gr.orientation 1               % portraid mode

! We'll create 9 different colored tiles, each 100 pixels across, laid
! out in a square grid.  The first row of 3 tiles will be placed at height
! 0-100, the next row at height 100-200, and the last row at height
! 200-300:
```

```
! 1st tile (rct1):

gr.color 255, 0, 0, 255, 1        % COLOR (transp, red, green, blue, fill)
gr.rect rct1, 0, 0, 100, 100      % POSITION (left, top, right bottom)

! 2nd tile (rct2):

gr.color 255, 0, 255, 0, 1        % (change color for every new box...)
gr.rect rct2, 100, 0, 200, 100    % (change position for every new box...)

! 3rd tile (rct3):

gr.color 255, 255, 0 , 0, 1       % color
gr.rect rct3, 200, 0, 300, 100    % create and position

! 4th tile:

gr.color 255, 255, 255, 0, 1      % color
gr.rect rct4, 0, 100, 100, 200    % position

! 5th tile:

gr.color 255, 0, 255, 255, 1      % color
gr.rect rct5, 100, 100, 200, 200  % position

! 6th tile:

gr.color 255, 128, 0, 0, 1        % color
gr.rect rct6, 200, 100, 300, 200  % position

! 7th tile:

gr.color 255, 0, 128, 0, 1        % color
gr.rect rct7, 0, 200, 100, 300    % position

! 8th tile:

gr.color 255, 0, 0, 0, 1          % color
gr.rect rct8, 100, 200, 200, 300  % position

! 9th tile (this one represents an empty space, so it's color is white):

gr.color 255, 255, 255, 255, 1    % color
gr.rect rct9, 200, 200, 300, 300  % position

! Show the screen

gr.render

! Now start an endless loop to constantly get user input, and then do
! something with that input:

getInput:

    ! Get a touch event:

    do
        gr.touch touched, x, y
    until touched

    ! Set the X and Y coordinate values to the CORNER of the touched
    ! tile's position (we want to deal with the specific position values
    ! of the touched tile, NOT with the random position at which the
    ! user's finger touched, somewhere within the tile):

    if x < 100          % if X is somewhere between 0-100, set X to 0
        x = 0
    elseif x < 200      % bewtween 101-200, set X to 100
        x = 100
    else                % bewtween 201-300, set X to 200
        x = 200
    endif
    if y < 100          % Do the same thing with all potential Y values
        y = 0
    elseif y < 200
        y = 100
    else
        y = 200
    endif

    ! Now we need to get the current positions of each tile, to determine
    ! which one was touched:

    gr.get.position rct1, x1, y1
    gr.get.position rct2, x2, y2
    gr.get.position rct3, x3, y3
    gr.get.position rct4, x4, y4
    gr.get.position rct5, x5, y5
    gr.get.position rct6, x6, y6
    gr.get.position rct7, x7, y7
    gr.get.position rct8, x8, y8
    gr.get.position rct9, x9, y9

    ! Compare each of the positions above to the touched corner
    ! coordinate, and marked it as the touched tile:

    if (x = x1) & (y = y1) then tilenum = rct1
    if (x = x2) & (y = y2) then tilenum = rct2
    if (x = x3) & (y = y3) then tilenum = rct3
    if (x = x4) & (y = y4) then tilenum = rct4
    if (x = x5) & (y = y5) then tilenum = rct5
    if (x = x6) & (y = y6) then tilenum = rct6
    if (x = x7) & (y = y7) then tilenum = rct7
    if (x = x8) & (y = y8) then tilenum = rct8
    if (x = x9) & (y = y9) then tilenum = rct9

    ! Get the position of tile #9 (the "blank" space):

    gr.get.position rct9, blankx, blanky

    ! Swap the position of the touched piece, and the blank piece.
    ! This effectively moves the touched piece into the blank space:

    gr.modify tilenum, "left", blankx           % move the touched piece
    gr.modify tilenum, "top", blanky            % to the former position
    gr.modify tilenum, "right", blankx + 100    % of the blank piece
    gr.modify tilenum, "bottom", blanky + 100
    gr.modify rct9, "left", x                   % move the black piece
    gr.modify rct9, "top", y                    % to the former position
    gr.modify rct9, "right", x + 100            % of the touched piece
    gr.modify rct9, "bottom", y + 100
    gr.render

! Do the entire process again and again, until the puzzle is solved:
```

```
goto getInput
```

Here's the complete program without comments:

```
gr.open 255, 255, 255, 255
gr.orientation 1
gr.color 255, 0, 0, 255, 1
gr.rect rct1, 0, 0, 100, 100
gr.color 255, 0, 255, 0, 1
gr.rect rct2, 100, 0, 200, 100
gr.color 255, 255, 0 , 0, 1
gr.rect rct3, 200, 0, 300, 100
gr.color 255, 255, 255, 0, 1
gr.rect rct4, 0, 100, 100, 200
gr.color 255, 0, 255, 255, 1
gr.rect rct5, 100, 100, 200, 200
gr.color 255, 128, 0, 0, 1
gr.rect rct6, 200, 100, 300, 200
gr.color 255, 0, 128, 0, 1
gr.rect rct7, 0, 200, 100, 300
gr.color 255, 0, 0, 0, 1
gr.rect rct8, 100, 200, 200, 300
gr.color 255, 255, 255, 255, 1
gr.rect rct9, 200, 200, 300, 300
gr.render
getInput:
    do
        gr.touch touched, x, y
    until touched
    if x < 100
        x = 0
    elseif x < 200
        x = 100
    else
        x = 200
    endif
    if y < 100
        y = 0
    elseif y < 200
        y = 100
    else
        y = 200
    endif
    gr.get.position rct1, x1, y1
    gr.get.position rct2, x2, y2
    gr.get.position rct3, x3, y3
    gr.get.position rct4, x4, y4
    gr.get.position rct5, x5, y5
    gr.get.position rct6, x6, y6
    gr.get.position rct7, x7, y7
    gr.get.position rct8, x8, y8
    gr.get.position rct9, x9, y9
    if (x = x1) & (y = y1) then tilenum = rct1
    if (x = x2) & (y = y2) then tilenum = rct2
    if (x = x3) & (y = y3) then tilenum = rct3
    if (x = x4) & (y = y4) then tilenum = rct4
    if (x = x5) & (y = y5) then tilenum = rct5
    if (x = x6) & (y = y6) then tilenum = rct6
    if (x = x7) & (y = y7) then tilenum = rct7
    if (x = x8) & (y = y8) then tilenum = rct8
    if (x = x9) & (y = y9) then tilenum = rct9
    gr.get.position rct9, blankx, blanky
    gr.modify tilenum, "left", blankx
    gr.modify tilenum, "top", blanky
    gr.modify tilenum, "right", blankx + 100
    gr.modify tilenum, "bottom", blanky + 100
    gr.modify rct9, "left", x
    gr.modify rct9, "top", y
    gr.modify rct9, "right", x + 100
    gr.modify rct9, "bottom", y + 100
    gr.render
goto getInput
```

11.14 Adding More Features to the Sliding Puzzle Game

The previous tile game provides a nice minimal example of how to respond to user touch, and how to move graphics interactively on screen. In order to make a more playable game, a few additional features are essential. First, keeping track of the number of swap moves provides a means of keeping score:

```
score = 0
gr.open 255, 255, 255, 255
gr.orientation 1
gr.color 255, 0, 0, 255, 1
gr.rect rct1, 0, 0, 100, 100
gr.color 255, 0, 255, 0, 1
gr.rect rct2, 100, 0, 200, 100
gr.color 255, 255, 0 , 0, 1
gr.rect rct3, 200, 0, 300, 100
gr.color 255, 255, 255, 0, 1
gr.rect rct4, 0, 100, 100, 200
gr.color 255, 0, 255, 255, 1
gr.rect rct5, 100, 100, 200, 200
gr.color 255, 128, 0, 0, 1
gr.rect rct6, 200, 100, 300, 200
gr.color 255, 0, 128, 0, 1
gr.rect rct7, 0, 200, 100, 300
gr.color 255, 0, 0, 0, 1
gr.rect rct8, 100, 200, 200, 300
gr.color 255, 255, 255, 255, 1
gr.rect rct9, 200, 200, 300, 300
gr.color 255, 0, 0, 0, 1
gr.text.draw scoreboard, 0, 350, "Move: " + str$(score)
gr.render
getInput:
    do
        gr.touch touched, x, y
    until touched
    if x < 100
        x = 0
    elseif x < 200
        x = 100
    else
        x = 200
    endif
    if y < 100
        y = 0
    elseif y < 200
        y = 100
    else
```

```
        y = 200
    endif
    gr.get.position rct1, x1, y1
    gr.get.position rct2, x2, y2
    gr.get.position rct3, x3, y3
    gr.get.position rct4, x4, y4
    gr.get.position rct5, x5, y5
    gr.get.position rct6, x6, y6
    gr.get.position rct7, x7, y7
    gr.get.position rct8, x8, y8
    gr.get.position rct9, x9, y9
    if (x = x1) & (y = y1) then tilenum = rct1
    if (x = x2) & (y = y2) then tilenum = rct2
    if (x = x3) & (y = y3) then tilenum = rct3
    if (x = x4) & (y = y4) then tilenum = rct4
    if (x = x5) & (y = y5) then tilenum = rct5
    if (x = x6) & (y = y6) then tilenum = rct6
    if (x = x7) & (y = y7) then tilenum = rct7
    if (x = x8) & (y = y8) then tilenum = rct8
    if (x = x9) & (y = y9) then tilenum = rct9
    gr.get.position rct9, blankx, blanky
    gr.modify tilenum, "left", blankx
    gr.modify tilenum, "top", blanky
    gr.modify tilenum, "right", blankx + 100
    gr.modify tilenum, "bottom", blanky + 100
    gr.modify rct9, "left", x
    gr.modify rct9, "top", y
    gr.modify rct9, "right", x + 100
    gr.modify rct9, "bottom", y + 100
    score = score + 1
    gr.modify scoreboard, "text", "Moves: " + str$(score)
    gr.render
goto getInput
```

You may have noticed that the above game allows for illegal moves (i.e., pieces that are not next to the blank space can be moved directly into the blank space). This next version compares the positions of the blank piece and the touched piece to eliminate that possibility:

```
score = 0
gr.open 255, 255, 255, 255
gr.orientation 1
gr.color 255, 0, 0, 255, 1
gr.rect rct1, 0, 0, 100, 100
gr.color 255, 0, 255, 0, 1
gr.rect rct2, 100, 0, 200, 100
gr.color 255, 255, 0 , 0, 1
gr.rect rct3, 200, 0, 300, 100
gr.color 255, 255, 255, 0, 1
gr.rect rct4, 0, 100, 100, 200
gr.color 255, 0, 255, 255, 1
gr.rect rct5, 100, 100, 200, 200
gr.color 255, 128, 0, 0, 1
gr.rect rct6, 200, 100, 300, 200
gr.color 255, 0, 128, 0, 1
gr.rect rct7, 0, 200, 100, 300
gr.color 255, 0, 0, 0, 1
gr.rect rct8, 100, 200, 200, 300
gr.color 255, 255, 255, 255, 1
gr.rect rct9, 200, 200, 300, 300
gr.color 255, 0, 0, 0, 1
gr.text.draw scoreboard, 0, 350, "Move: " + str$(score)
gr.render
getInput:
    do
        gr.touch touched, x, y
    until touched
    if x < 100
        x = 0
    elseif x < 200
        x = 100
    else
        x = 200
    endif
    if y < 100
        y = 0
    elseif y < 200
        y = 100
    else
        y = 200
    endif
    gr.get.position rct1, x1, y1
    gr.get.position rct2, x2, y2
    gr.get.position rct3, x3, y3
    gr.get.position rct4, x4, y4
    gr.get.position rct5, x5, y5
    gr.get.position rct6, x6, y6
    gr.get.position rct7, x7, y7
    gr.get.position rct8, x8, y8
    gr.get.position rct9, x9, y9
    if (x = x1) & (y = y1) then tilenum = rct1
    if (x = x2) & (y = y2) then tilenum = rct2
    if (x = x3) & (y = y3) then tilenum = rct3
    if (x = x4) & (y = y4) then tilenum = rct4
    if (x = x5) & (y = y5) then tilenum = rct5
    if (x = x6) & (y = y6) then tilenum = rct6
    if (x = x7) & (y = y7) then tilenum = rct7
    if (x = x8) & (y = y8) then tilenum = rct8
    if (x = x9) & (y = y9) then tilenum = rct9
    gr.get.position rct9, blankx, blanky
    diffx = abs(x - blankx)
    diffy = abs(y - blanky)
    if ((diffx + diffy) < 101)
        gr.modify tilenum, "left", blankx
        gr.modify tilenum, "top", blanky
        gr.modify tilenum, "right", blankx + 100
        gr.modify tilenum, "bottom", blanky + 100
        gr.modify rct9, "left", x
        gr.modify rct9, "top", y
        gr.modify rct9, "right", x + 100
        gr.modify rct9, "bottom", y + 100
        score = score + 1
        gr.modify scoreboard, "text", "Moves: " + str$(score)
        gr.render
    endif
goto getInput
```

Finally, adding some text to each tile makes it easier to clarify the order in which the tiles should be rearranged. In the following example, numbers are added in a text field, placed at the center of each tile. The appropriate text objects are moved whenever their associated tile images are moved:

```
score = 0
gr.open 255, 255, 255, 255
gr.orientation 1
gr.color 255, 0, 0, 255, 1
gr.rect rct1, 0, 0, 100, 100
gr.color 255, 0, 255, 0, 1
gr.rect rct2, 100, 0, 200, 100
gr.color 255, 255, 0 , 0, 1
gr.rect rct3, 200, 0, 300, 100
gr.color 255, 255, 255, 0, 1
gr.rect rct4, 0, 100, 100, 200
gr.color 255, 0, 255, 255, 1
gr.rect rct5, 100, 100, 200, 200
gr.color 255, 128, 0, 0, 1
gr.rect rct6, 200, 100, 300, 200
gr.color 255, 0, 128, 0, 1
gr.rect rct7, 0, 200, 100, 300
gr.color 255, 0, 0, 0, 1
gr.rect rct8, 100, 200, 200, 300
gr.color 255, 255, 255, 255, 1
gr.rect rct9, 200, 200, 300, 300
gr.color 255, 0, 0, 0, 1
gr.text.draw scoreboard, 0, 350, "Move: " + str$(score)
gr.color 255, 255, 255, 255, 1
gr.text.draw text1, 50, 50, "1"
gr.text.draw text2, 150, 50, "2"
gr.text.draw text3, 250, 50, "3"
gr.text.draw text4, 50, 150, "4"
gr.text.draw text5, 150, 150, "5"
gr.text.draw text6, 250, 150, "6"
gr.text.draw text7, 50, 250, "7"
gr.text.draw text8, 150, 250, "8"
gr.text.draw text9, 250, 250, "9"
gr.render
getInput:
    do
        gr.touch touched, x, y
    until touched
    if x < 100
        x = 0
    elseif x < 200
        x = 100
    else
        x = 200
    endif
    if y < 100
        y = 0
    elseif y < 200
        y = 100
    else
        y = 200
    endif
    gr.get.position rct1, x1, y1
    gr.get.position rct2, x2, y2
    gr.get.position rct3, x3, y3
    gr.get.position rct4, x4, y4
    gr.get.position rct5, x5, y5
    gr.get.position rct6, x6, y6
    gr.get.position rct7, x7, y7
    gr.get.position rct8, x8, y8
    gr.get.position rct9, x9, y9
    if (x = x1) & (y = y1)
        tilenum = rct1
        textnum = text1
    endif
    if (x = x2) & (y = y2)
        tilenum = rct2
        textnum = text2
    endif
    if (x = x3) & (y = y3)
        tilenum = rct3
        textnum = text3
    endif
    if (x = x4) & (y = y4)
        tilenum = rct4
        textnum = text4
    endif
    if (x = x5) & (y = y5)
        tilenum = rct5
        textnum = text5
    endif
    if (x = x6) & (y = y6)
        tilenum = rct6
        textnum = text6
    endif
    if (x = x7) & (y = y7)
        tilenum = rct7
        textnum = text7
    endif
    if (x = x8) & (y = y8)
        tilenum = rct8
        textnum = text8
    endif
    if (x = x9) & (y = y9)
        tilenum = rct9
        textnum = text9
    endif
    gr.get.position rct9, blankx, blanky
    diffx = abs(x - blankx)
    diffy = abs(y - blanky)
    if ((diffx + diffy) < 101)
        gr.modify tilenum, "left", blankx
        gr.modify tilenum, "top", blanky
        gr.modify tilenum, "right", blankx + 100
        gr.modify tilenum, "bottom", blanky + 100
        gr.modify textnum, "x", blankx + 50
        gr.modify textnum, "y", blanky + 50
        gr.modify rct9, "left", x
        gr.modify rct9, "top", y
        gr.modify rct9, "right", x + 100
        gr.modify rct9, "bottom", y + 100
        score = score + 1
        gr.modify scoreboard, "text", "Moves: " + str$(score)
        gr.render
    endif
goto getInput
```

11.15 Scaling Graphics to Fit Different Screen Resolutions

RFO Basic includes a simple feature to scale graphics to fit properly on screens of any resolution. The following code, taken directly from the De Re Basic Manual demonstrates how to scale your screens. All you need to do is set the initial 2 values (di_height and di_width), and the entire graphic application will be scaled in relation to those set size, no matter how big or how small the screen dimensions are:

```
di_height = 480
di_width = 800
gr.open 255, 255, 255, 255
gr.orientation 0
gr.screen actual_w, actual_h
scale_width = actual_w / di_width
scale_height = actual_h / di_height
gr.scale scale_width, scale_height
```

12. More Graphic Apps

12.1 Catch Game

Catch is a simple game in which red circle objects drop from the sky. The falling circles need to be caught before hitting the ground. The player image is represented by a blue box on the bottom of the screen. The user touches the left and right hand sides of the screen to move the player piece side to side. One point is added every time a falling red circle is caught by (collides with) the blue player box. 1 point is subtracted every time a falling red circle makes it to the bottom of the screen without being caught. The game speed increases progressively with time. Press the back key any time to end the game. Game play continues only while the user touches the screen. The goal is to end when you think you have the best possible score.

You'll notice that most of the code and the outline of this program is similar to the sliding tile puzzle app. The gr_collision() function is used to determine if the red circle is touching the blue box. Whenever the red circle is caught or reaches the bottom of the screen, it is repositioned to a random coordinate above the top of the screen and dropped again.

```
score = 0
speed = 4.0
gr.open 255, 255, 255, 255
gr.orientation 1
gr.color 255, 30, 30, 30, 1     % play area
gr.rect area, 0, 300, 0, 300    % play area
gr.color 255, 255, 0, 0, 1
gr.circle ball, round(rnd() * 300), 0, 10
gr.color 255, 0, 0, 255, 1
gr.rect player, 150, 280, 200, 300
gr.text.draw scoreboard, 0, 350, "Score: " + str$(score)
gr.render
do
    do
        gr.touch touched, x, y
    until touched
    gr.get.position ball, x1, y1
    gr.get.position player, x2, y2
    if x < (x2 + 15)
        let direction = -10
    elseif x > (x2 + 35)
        let direction = 10
    else
        let direction = 0
    endif
    gr.modify ball, "y", y1 + round(speed)
    gr.modify player, "left", x2 + direction
    gr.modify player, "right", x2 + 50 + direction
    gr.render
    if gr_collision(ball, player)
        score = score + 1
        speed = speed + 0.2
        gosub positionBall
    endif
    if y1 > 300
        score = score - 1
        gosub positionBall
    endif
    gr.render
    pause 5
until 1 = 2
positionBall:
    gr.modify scoreboard, "text", "Score: " + str$(score)
    gr.modify ball, "y", 0
    gr.modify ball, "x", round(rnd() * 300)
return
OnBackKey:
    popup "Score:  " + str$(score),0,0,1
    end
```

12.2 Calculator

Every programming tutorial includes av requisite calculator example. Here's one for RFO Basic:

```
! Start by assigning some initial variables, to be used later:

prevVal = 0
curVal = 0
displayFlag = 0

! Open the graphics screen and set initial display parameters:

gr.open 255,248,248,248
gr.color 255,0,0,0,0
gr.orientation 1
gr.text.bold 1

! Draw a rectangle on the screen to be used as the number display:

gr.rect displayBox,10,10,300,50

! Draw some text on the screen, which will display the numbers typed by
! the user and the result of any calculation:

gr.text.draw displayText,20,35,"0"

! This whole next section diplays the buttons on the screen.  It would
! have been a bit more straightforward to simply draw each rectangle and
! text manually.  Instead, an array holding the text and coordinates of
! each button is first created, and then a FOR loop is used to position
! each item on screen.  This allows for easy repositioning and resizing
! for different screen dimensions, and other future layout changes/
! features:

! This array holds the text and coordinates of each button:

array.load buttonData$[],~
    "1","30","60","2","90","60","3","150","60",~
    "4","30","120","5","90","120","6","150","120",~
    "7","30","180","8","90","180","9","150","180",~
    "0","30","240","+","90","240","-","150","240",~
    "*","30","300","/","90","300","=","150","300"

! This line sets the variable to the number of items in the above list.
! This allows more buttons to be added later, without having to manually
! edit any variables.:

array.length numButtons,buttonData$[]

! Create 2 new empty arrays that will hold references to each rectangle
! and text item that makes up each graphical button:

dim rectNames[numButtons]
dim textNames[numbuttons]

! Set some variables containing the size and horizontal/vertical offset of
! all buttons (these values could easily by calculated as products of the
! width and height of the screen):

bs = 20         % button size
xshift = 65     % this moves all buttons over a given # of pixels
yshift = 40     % this moves all buttons down a given # of pixels

! This loop creates each button by running though the buttonData$[] array
! above, pulling out groups of 3 items from the list:

for i = 1 to numButtons step 3

    ! Set the left and top positions for each consecutive text item:

    lPos = val(buttonData$[i+1]) + xshift
    tPos = val(buttonData$[i+2]) + yshift

    ! Draw a blue rectangle around the location of each text item.
    ! Notice that the positions of the rectangle are based on the text
    ! positions, the button size variable (bs), and ten extra pixels of
    ! offset on the top and right sides, to accomodate the size of the
    ! text (this offset could also be calculated automatically):

    gr.color 255,0,0,255,1
    gr.rect rectNames[i],lPos-bs,tPos-bs-10,lPos+bs+10,tPos+bs

    ! Draw the text item in white:

    gr.color 255,255,255,255,1
    gr.text.draw textNames[i],lPos,tPos,buttonData$[i]

next i

! Set the color back to black and show the screen:

gr.color 255,0,0,0,0
gr.render

! This endless loop waits for user input:

getInput:

    ! Wait for text input and save the coordinates in the variables "x"
    ! and "y":
```

```
    do
        gr.touch touched,x,y
    until touched

    ! This FOR loop runs through the buttonData$[] array every time the
    ! user touches the screen, to check which button has been pressed,
    ! and responds with appropriate action(s):

    for i = 1 to numbuttons step 3

        ! Each consecutive time through the loop, the coordinates of each
        ! button text are evaluated:

        lPos = val(buttonData$[i+1]) + xshift
        tPos = val(buttonData$[i+2]) + yshift

        ! The location of the touch is compared with the location of each
        ! button in the list:

        if x > (lpos-bs) & x < (lpos+bs+10) & ~
                y > (tpos-bs-10) & y < (tpos+bs)

            ! If the touched occured within the rectangle surrounding the
            ! current button text, set the variable q$ to the matching
            ! button's text:

            q$ = buttonData$[i]

            ! Number buttons behave differently than operator buttons:

            if q$ = "1" | q$ = "2" | q$ = "3" | q$ = "4" | q$ = "5" |~
                    q$ = "6" | q$ = "7" | q$ = "8" | q$ = "9" | q$ = "0"

                ! This flag is used to update the screen properly (the
                ! display must be cleared if a total is currently
                ! showing - this flag is used to track that state):

                if displayFlag = 1
                    gr.modify displayText,"text",display$
                    gr.render
                    displayFlag = 0
                endif

                ! If a "0" was on the screen, clear and update the
                ! display:

                if display$ = "0"
                    display$ = ""
                    gr.modify displayText,"text",display$
                    gr.render
                endif

                ! Append the text of the selected button to the current
                ! display text, and update the screen:

                display$ = display$ + buttonData$[i]
                gr.modify displayText,"text",display$
                gr.render

                ! Set the variable "curVal" to hold the numerical value
                ! of the text currently displayed on screen:
```

```
        curVal = val(display$)

    ! All of the operator keys perform the same way.  They set
    ! the variable "operator$" to the operation the key performs,
    ! and then they run the "setEval" subroutine:

    elseif q$ = "+"
        operator$ = "+"
        gosub setEval
    elseif q$ = "-"
        operator$ = "-"
        gosub setEval
    elseif q$ = "*"
        operator$ = "*"
        gosub setEval
    elseif q$ = "/"
        operator$ = "/"
        gosub setEval

    ! The equal key performs all the following actions:

    elseif q$ = "="

        ! Check to see if the displayFlag variable is in the
        ! appropriate state (the total is not being displayed):

        if displayFlag = 0

            ! If the user tries to divide by zero, pop up an error
            ! message and exit the current loop:

            if operator$ = "/" & curVal = 0.0
                popup "Division by 0 is not allowed.",0,0,1
                goto getInput
            endif

            ! Perform the appropriate math evaluation, based upon
            ! the current state of the operator$ variable (set
            ! above, whenever the user clicks on an operator key):

            if operator$ = "+"
                curVal = curVal + prevVal
            elseif operator$ = "-"
                curVal = prevVal - curVal
            elseif operator$ = "*"
                curVal = curVal * prevVal
            elseif operator$ = "/"
                curVal = prevVal / curVal
            endif

            ! Convert the updated "curVal" value to a string, and
            ! update the display:

            display$ = str$(curVal)
            gr.modify displayText,"text",display$
            gr.render

            ! Set the displayFlag variable to indicate that a
            ! total is currently being displayed on the screen:

            displayFlag = 1
```

```
                endif
            endif
        endif
    next i

! Now go back and wait for more user input, endlessly:

goto getInput

! This is the subroutine run when any of the operator keys are pressed:

setEval:
    ! Save the value currently on string in the variable "prevVal",
    ! to be used later:

    prevVal = curVal

    ! Clear the display text, and update the screen:

    display$ = ""
    gr.modify displayText,"text",display$
    gr.render

return

! This last bit of code ends the program gracefully when the user clicks
! the Android back button:

onError:
    cls
    end
```

Here's the whole program without comments:

```
prevVal = 0
curVal = 0
displayFlag = 0

gr.open 255,248,248,248
gr.color 255,0,0,0,0
gr.orientation 1
gr.text.bold 1

gr.rect displayBox,10,10,300,50
gr.text.draw displayText,20,35,"0"

array.load buttonData$[],~
    "1","30","60","2","90","60","3","150","60",~
    "4","30","120","5","90","120","6","150","120",~
    "7","30","180","8","90","180","9","150","180",~
    "0","30","240","+","90","240","-","150","240",~
    "*","30","300","/","90","300","=","150","300"
array.length numButtons,buttonData$[]
dim rectNames[numButtons]
dim textNames[numbuttons]
bs = 20         % button size
xshift = 65     % this moves all buttons over a given # of pixels
yshift = 40     % this moves all buttons down a given # of pixels
for i = 1 to numButtons step 3
    lPos = val(buttonData$[i+1]) + xshift
    tPos = val(buttonData$[i+2]) + yshift
    gr.color 255,0,0,255,1
    gr.rect rectNames[i],lPos-bs,tPos-bs-10,lPos+bs+10,tPos+bs
    gr.color 255,255,255,255,1
    gr.text.draw textNames[i],lPos,tPos,buttonData$[i]
next i

gr.color 255,0,0,0,0
gr.render

getInput:
    do
        gr.touch touched,x,y
    until touched
    for i = 1 to numbuttons step 3
        lPos = val(buttonData$[i+1]) + xshift
        tPos = val(buttonData$[i+2]) + yshift
        if x > (lpos-bs) & x < (lpos+bs+10) & ~
                y > (tpos-bs-10) & y < (tpos+bs)
            q$ = buttonData$[i]
            if q$ = "1" | q$ = "2" | q$ = "3" | q$ = "4" | q$ = "5" |~
                    q$ = "6" | q$ = "7" | q$ = "8" | q$ = "9" | q$ = "0"
                if displayFlag = 1
                    gr.modify displayText,"text",display$
                    gr.render
                    displayFlag = 0
                endif
                if display$ = "0"
                    display$ = ""
                    gr.modify displayText,"text",display$
                    gr.render
                endif
                display$ = display$ + buttonData$[i]
                gr.modify displayText,"text",display$
                gr.render
                curVal = val(display$)
            elseif q$ = "+"
                operator$ = "+"
                gosub setEval
            elseif q$ = "-"
                operator$ = "-"
                gosub setEval
            elseif q$ = "*"
                operator$ = "*"
                gosub setEval
            elseif q$ = "/"
                operator$ = "/"
                gosub setEval
            elseif q$ = "="
                if displayFlag = 0
                    if operator$ = "/" & curVal = 0.0
                        popup "Division by 0 is not allowed.",0,0,1
                        goto getInput
                    endif
                    if operator$ = "+"
                        curVal = curVal + prevVal
                    elseif operator$ = "-"
                        curVal = prevVal - curVal
                    elseif operator$ = "*"
                        curVal = curVal * prevVal
                    elseif operator$ = "/"
                        curVal = prevVal / curVal
```

```
                      endif
                      display$ = str$(curVal)
                      gr.modify displayText,"text",display$
                      gr.render
                      displayFlag = 1
                  endif
              endif
          endif
      next i
goto getInput
setEval:
    prevVal = curVal
    display$ = ""
    gr.modify displayText,"text",display$
    gr.render
return
onError:
    cls
    end
```

12.3 Snake

12.4 Ski

12.5 Simple Shoot-Em-Up Game

12.6 Guitar Chord Diagram Maker

12.7 Image Viewer With Swipe Gestures

12.8 Thumbnail Image Maker

13. Building GUI Interfaces Using Graphics

13.1 Basics - No HTML Required

You've already seen how to create GUI interfaces using HTML. In cases where graphics and animation need to be mixed with GUIs, or in situations where you prefer not to use the HTML/Javascript interface, it can be useful to build GUI interfaces from scratch using native graphic elements. The following example demonstrates some basic techniques for creating interactive GUIs using RFO Basic commands:

```
! This example will allow the user to input 4 pieces of data.  Set those
! variables initially to null:

text$ = ""
bigtext$ = ""
number = 0
listselection$ = ""

! Open the graphics screen:

gr.open 255,255,255,255
gr.color 255,0,0,0,0
gr.orientation 1

! Draw a text label on screen and a text area where the entered text will
! be displayed.  Underline the text area.  This text area will accept
! simple text (entered via an "input" function):
```

```
gr.text.draw t1,10,50,"Text:"
gr.text.draw f1,100,50,""
gr.line l1, 100,52,300,52

! Draw another label, text area and underline.  This will accept
! long text (entered via a "text.input" function):

gr.text.draw t2,10,100,"Big Text:"
gr.text.draw f2,100,100,""
gr.line l2, 100,102,300,102

! Another label, area, and underline.  This accepts numbers (entered via
! an "input" function, with the input variable type set to a number,
! instead of a string):

gr.text.draw t3,10,150,"Number:"
gr.text.draw f3,100,150,""
gr.line l3, 100,152,300,152

! Another label, area, and underline.  This accepts data selected by the
! user from a selection list (entered via a "select" function):

gr.text.draw t4,10,200,"List Select:"
gr.text.draw f4,100,200,""
gr.line l4, 100,202,300,202

! Add a red "Submit" link which the user will click after all data has
! been entered:

gr.color 255,255,0,0,0
gr.text.draw b1,100,270,"SUBMIT"
gr.color 255,0,0,0,0

! Show the screen:

gr.render

! Start an endless loop that accepts user input:

getText:

    ! Wait for the user to touch the screen.  Assign the touched
    ! coordinate to the variables "x" and "y":

    do
        gr.touch touched, x, y
    until touched

    ! Hide the graphics screen momentarily (to show input dialogs):

    gr.front 0

    ! Respond appropriately, according the position at which the screen
    ! was touched:

    if y < 75

        ! "Text" field touched.  Run the "input" function and update the
        ! text field with the user entered results:

        input "Text:", text$, text$
        gr.modify f1, "text", text$

    elseif y < 125

        ! "Big Text" field touched.  Run the "text.input" function and
        ! update the text field with the user entered results:

        text.input bigtext$, bigtext$
        gr.modify f2, "text", bigtext$

    elseif y < 175

        ! "Number" field touched.  Run the "input" function and update the
        ! text field with the user entered results (formatted as needed):

        input "Number:", number, number
        gr.modify f3, "text", replace$(str$(number), ".0", "")

    elseif y < 225

        ! "List Select" field touched.  Run the "selected" function and
        ! update the text field with the user selected results:

        array.delete list$[]
        array.load list$[], "Red", "Green", "Blue"
        select selecteditem, list$[], "Favorite Color:"
        listselection$ = list$[selecteditem]
        gr.modify f4, "text", listselection$

    elseif y > 250 & y < 300

        ! "Submit" link touched.  In your program, this is where you will
        ! procees the data submitted by the user.  In this example, the
        ! entered data is simply printed on the screen:

        gr.front 0
        print "Submitted:\n\n"; text$,bigtext$,number,listselection$
        pause 2000

    endif

    ! Show the graphics screen update to display the entered data:

    gr.front 1
    gr.render

! Continue to accept input until the user submits data or exits:

goto getText

! If the user exits the program with the back button, exit gracefully:

onError:
    cls
    end
```

Here's the whole program without comments:

```
text$ = ""
bigtext$ = ""
```

```
number = 0
listselection$ = ""

gr.open 255,255,255,255
gr.color 255,0,0,0,0
gr.orientation 1

gr.text.draw t1,10,50,"Text:"
gr.text.draw f1,100,50,""
gr.line l1, 100,52,300,52

gr.text.draw t2,10,100,"Big Text:"
gr.text.draw f2,100,100,""
gr.line l2, 100,102,300,102

gr.text.draw t3,10,150,"Number:"
gr.text.draw f3,100,150,""
gr.line l3, 100,152,300,152

gr.text.draw t4,10,200,"List Select:"
gr.text.draw f4,100,200,""
gr.line l4, 100,202,300,202

gr.color 255,255,0,0,0
gr.text.draw b1,100,270,"SUBMIT"
gr.color 255,0,0,0,0

gr.render

getText:
    do
        gr.touch touched, x, y
    until touched
    gr.front 0
    if y < 75
        input "Text:", text$, text$
        gr.modify f1, "text", text$
    elseif y < 125
        text.input bigtext$, bigtext$
        gr.modify f2, "text", bigtext$
    elseif y < 175
        input "Number:", number, number
        gr.modify f3, "text", replace$(str$(number), ".0", "")
    elseif y < 225
        array.delete list$[]
        array.load list$[], "Red", "Green", "Blue"
        select selecteditem, list$[], "Favorite Color:"
        listselection$ = list$[selecteditem]
        gr.modify f4, "text", listselection$
    elseif y > 250 & y < 300
        gr.front 0
        print "Submitted:\n\n"; text$,bigtext$,number,listselection$
        pause 2000
    endif
    gr.front 1
    gr.render
goto getText
onError:
    cls
    end
```

13.2 A Typical GUI Form With a Few More Bells and Whistles

This example demonstrates how to add a rectangular button that resizes to fit the button text, rectangular areas for text input, and right justified text labels:

```
text$ = ""
bigtext$ = ""
number = 0
listselection$ = ""

gr.open 255,248,248,248
gr.color 255,0,0,0,0
gr.orientation 1

gr.text.align 3
gr.text.draw t1,75,50,"Text:"
gr.text.align 1
gr.text.draw f1,100,50,""
gr.rect l1, 95,27,300,65

gr.text.align 3
gr.text.draw t2,75,100,"Big Text:"
gr.text.align 1
gr.text.draw f2,100,100,""
gr.rect l2, 95,77,300,115

gr.text.align 3
gr.text.draw t3,75,150,"Number:"
gr.text.align 1
gr.text.draw f3,100,150,""
gr.rect l3, 95,127,300,165

gr.text.align 3
gr.text.draw t4,75,200,"List Select:"
gr.text.align 1
gr.text.draw f4,100,200,""
gr.rect l4, 95,177,300,215

gr.text.bold 1
btn1$ = "SUBMIT"
gr.color 255,0,0,0,1
gr.get.textbounds btn1$,l,t,r,b
gr.rect btn1,l+110-10,t+270-10,r+110+10,b+270+10     % match b1 pos (110)
gr.text.bold 0
gr.color 255,255,255,255,0
gr.text.draw b1,110,270,btn1$
gr.color 255,0,0,0,0

gr.render

getText:
    do
        gr.touch touched, x, y
    until touched
    gr.front 0
    if y < 75
        input "Text:", text$, text$
        gr.modify f1, "text", text$
    elseif y < 125
        text.input bigtext$, bigtext$
        gr.modify f2, "text", bigtext$
    elseif y < 175
        input "Number:", number, number
        gr.modify f3, "text", replace$(str$(number), ".0", "")
    elseif y < 225
        array.delete list$[]
        array.load list$[], "Red", "Green", "Blue"
        select selecteditem, list$[], "Favorite Color:"
        listselection$ = list$[selecteditem]
        gr.modify f4, "text", listselection$
    elseif y > 250 & y < 300
        gr.front 0
        print "Submitted:\n\n"; text$,bigtext$,number,listselection$
        pause 2000
    endif
    gr.front 1
    gr.render
goto getText
onError:
    cls
    end
```

---Recipe Database #3 (This Time With a Graphic GUI)

14. Transferrting Data Via Network Sockets

Computing devices connected together on the same wired or wireless network can transfer data and files directly to one another via network (TCP and UDP) "socket" connections.

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 computers connected to home routers are normally in the local IP range "192.168.xxx.xxx". You can obtain the IP address of your local computer with the following RFO Basic code:

(to be completed...)

14.1 Ports

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.

14.2 Servers and Clients

"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 RFO Basic functions 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:

(to be completed...)

Here's the CLIENT. Run it in a *separate* instance of the RFO Basic interpreter, after the above program has been started:

(to be completed...)

14.3 Blocking and Non-Blocking Loops

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):

(to be completed...)

Here's the client program. Run it only *after the server program has been started*, and in a separate instance of the RFO Basic interpreter (or on a separate computer):

(to be completed...)

It's important to understand that 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:

(to be completed...)

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 RFO Basic interpreter, and be sure to start the server first:

(to be completed...)

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:

(to be completed...)

The following script is a complete 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 read from and write to a file on a universally available *third party* FTP server):

(to be completed...)

Here's an even more compact version (probably the shortest instant messenger program you'll ever see!):

(to be completed...)

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:

(to be completed...)

14.4 Port Forwarding and VPNs

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).

14.4.1 Transferring Binary Files Through TCP Network Sockets:

(to be completed...)

14.5 Walkie Talkie (VOIP)

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 endless loops to enable continuous communication back and forth.

(to be completed...)

14.6 Desktop File Transfer

Here's a little app to transfer any selected file from desktop OS's to Android phone, over WIFI. The selected file can be any type of file: text or binary: image, sound, etc. On your Android run the following client code in RFO basic:

```
input "Save Filename", filename$, "test.jpg"
input "Connect-to IP", ip$,"192.168.1.136"
input "Port number", port, 2345
socket.client.connect ip$, port
print "Connected"
maxclock = clock() + 30000
do
    socket.client.read.ready flag
    if clock() > maxclock
        print "Process timed out. Ending."
        end
    endif
until flag
byte.open W, fw, filename$
socket.client.read.file fw
byte.close fw
```

```
socket.client.close
print "Done"
```

A downloadable .apk file of the above app is available at http://laughton.com/basic/programs/filetransfer.apk

The desktop server is written in REBOL. There are versions of REBOL for Windows, Mac, Linux, BSD, Solaris, QNX, AIX, HP-UX, Windows CE, BEOS, Amiga, etc., so you can use this code to transfer files between a bunch different OS's and Android. Just go to http://rebol.com and download the REBOL interpreter for your OS (it's only about 1/2 meg), and paste the following code into the console (or save it in a .r file):

```
rebol []
selected-file: request-file/only
port-num: request-text/title/default "Port number:" "2345"
print rejoin [
    "Server started on " (read join dns:// read dns://) ":" port-num
]
if error? try [
    port: first wait open/binary/no-wait join tcp://: port-num
] [quit]
file: read/binary selected-file
insert port file
close port
print "Done" halt
```

A Windows .exe is available at http://laughton.com/basic/programs/file_transfer.exe

15. SQLite Databases

One of the most powerful features of RFO Basic is its built-in support for SQLite. SQLite is a small, self contained relational database system that allows you to store, retrieve, search, sort, compare, and otherwise manipulate large stores of data, very quickly and easily. SQLite is used as the main data storage system in Android devices, and is supported on just about every other modern computing platform (PC, Mac, Linux, iPhone and other mobile devices, Webkit based browser apps, Web Servers, etc.). You can easily copy and transfer entire SQLite databases of information created on your Android, and work with them on devices with different operating systems, and visa-versa.

Keep in mind that programming, and the use of computing devices in general, is ultimately about managing *data*, so learning to use SQLite is fundamentally useful in becoming a capable RFO Basic programmer.

15.1 Tables

In SQLite 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
```

15.2 SQL

"SQL" statements let you work with data stored in database tables. 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, ...)       % add data
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 following SQL code will create the Contacts table illustrated above:

```
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')
```

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%'  % any name starting with "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
```

There are many tutorials available online to help learn the SQL language, and SQLite, in greater depth. Take a look at http://zetcode.com/databases/sqlitetutorial for more information.

To open an SQLite database, use the following code. The file name can be anything you choose, and all the other parameters are optional. If the database file does not exist, it will be created:

```
db = SqlOpenDatabase("myData.db", "1.0", "My Data")
If db = 0 Then MsgBox "Error opening database"
```

SQL commands should be formatted into an array. You can define your own functions to handle successful and failed SQL commands. The dbSuccess function below is very important. It demonstrates how to use a For loop to retreive and display data from an SQL SELECT command:

(to be completed...)

To execute the SQL commands, use the Sql function, with the database ID and SQL command array as parameters:

(to be completed...)

Here's a complete example that opens a database, creates a new "myData" table, inserts data into the table, and then retrieves and prints all the contents of the table:

(to be completed...)

16. Debugging

RFO Basic has a number of debug functions that you can use to help find and fix errors with your code. The debug functions can be used to print the contents of all existing scalar variables, arrays, lists, etc. You can place debug functions throughout your code, and then use the debug.on and debug.off functions to activate and deactivate all debugging activities at once. The debug.show.___ functions are especially useful. The debug.show.scalars function shows all existing variables and their values, and waits for user interaction before continuing the program:

```
debug.on
x = 1
y = 2
debug.show.scalars
z = 3
debug.show.scalars
end
```

There are similar debug.show.list, debug.show.array and other functions to show and watch all possible data structures in your code.

When writing graphic applications, the gr.front function must first be used to switch to console mode, then the debug functions can be run, then the gr.front and gr.render functions must be run to switch back to graphic mode:

```
gr.front 0
debug.on
debug.show.scalars
```

```
gr.front 1
gr.render
```

17. Strings

A "string" is simply a series of characters. Take a look at the following examples to see how to do a few common text operations:

(to be completed...)

18. Quick Review and Synopsis

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

1. To start off, RFO Basic has many built-in function words that perform common tasks. As in other languages, function words are typically followed by data parameters. Parameters are placed immediately after the function word and are typically separated by commas. To accomplish a desired goal, functions are arranged in succession, one after another. The value returned by one function is often used as the argument 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. You can complete significant work by simply knowing the predefined functions in the language, and organizing them into a useful order.
2. Empty white space (spaces, tabs, newlines, etc.) can be inserted as desired to make code more readable. The tilde character is used to continue code onto multiple lines. Text after a percent sign and before a new line is treated as a comment. Exclamation points comment out entire lines. Double exclamation points comment out multiple lines.
3. RFO Basic contains a rich set of conditional and looping structures, which can be used to manage program flow and data processing activities. If, while/repeat, do/until, for, and other typical structures are supported.
4. RFO Basic can increment, compare, and perform computations between items in lists, using FOR loops. Data of any type can be written to and read from files, to SQLite databases, or sent and retrieved to/from Internet servers, either by FTP, by http.post, and by direct socket connections.
5. Any data can be assigned a variable label. The equal sign ("=") is used to assign variable word labels to values. Many functions include the return variable immediately after the function word, in the function's parameter list. Once assigned, variable words can be used to represent all of the data and/or actions contained in the given string, array, etc.
6. Multiple pieces of data (lists) are stored in arrays, lists, and bundles. Items can be added to or picked out of lists by index numbers. You can concatenate lists into serialized strings and saved to file, and lists of data can also be saved to database tables. The "split" function can be used to convert strings back to to lists.
7. You can use graphics screen not only to display images and graphic designs, but also to present useful GUI layouts.

19. Built In Help and Documentation

20. Additional Topics

(under construction)

21. REAL WORLD CASE STUDIES - Learning To Think In Code

At this point, you've seen most essential bits of RFO Basic language syntax and API, 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 RFO Basic code, and how to organize your work process to approach any unique situation. Case studies are presented to provide insight as to how specific real life situations were satisfied.

21.1 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 devices. 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: 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 selection lists, text input fields, etc? 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, list structures and functions, etc? Will the data be stored in local files, in remote files, in a database, or just in temporary memory (variables), etc? 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 into functions, subroutines, etc? 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 user interactions, 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 interaction 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 "menu option (which does this when clicked...)", "list: (with labels and sub-lists 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. The API reference 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 RFO Basic's help documents. 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 selecting from a list of menu options. That should elicit the thought of certain bits of select function structure, and you can begin to write the code outline to implement those choices.

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 menu choice doing something for your user, you don't need to immediately write the RFO Basic code that the menu choice runs. Initially, just write a *description* of what you want the choice 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 your 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 labels, 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, menu selection, action, etc., to the next.

21.2 Case 1 - Scheduling Teachers

21.3 Case 2 - Days Between Two Dates Calculator

21.4 Case 3 - Simple Search

21.5 Case 4 - A Calculator

21.6 Case 5 - Backup Music Generator ("Jam Tool")

21.7 Case 6 - FTP Tool

21.8 Case 7 - Jeopardy

21.9 Case 8 - Tetris

21.10 Case 9 - Media Player

21.11 Case 10 - Web Site Spidering App

(the rest of this section is under construction)

22. Other Scripts

22.1 Number Verbalizer

22.2 Bingo

22.3 Voice Alarms