Liberty BASIC v1.4 Help Document

Here are the various topics in this on-line help file:

Overview of Liberty BASIC v1.4

Welcome to our Liberty BASIC overview.   In this chapter we will introduce you to:

The Liberty BASIC Editor
 This is the place where BASIC programs are written and compiled.

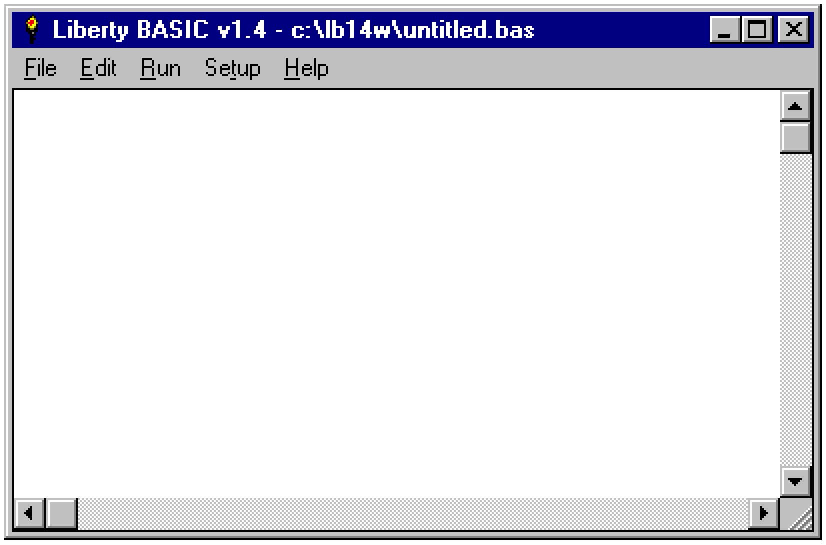
Writing your own Programs
 Getting started

Using the Debugger
 How to debug your Liberty BASIC programs

Creating a tokenized file
 Making your programs do more for you

The Liberty BASIC Editor:

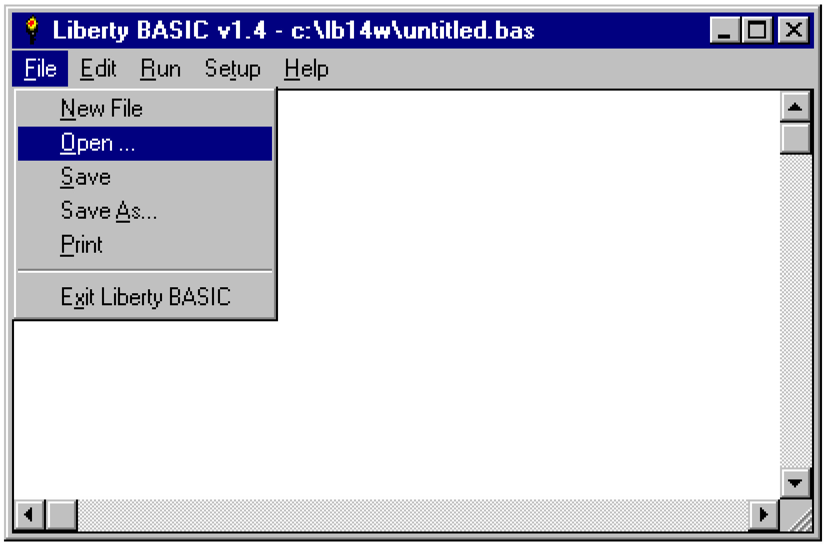
When you start Liberty BASIC, you will see a window like this:



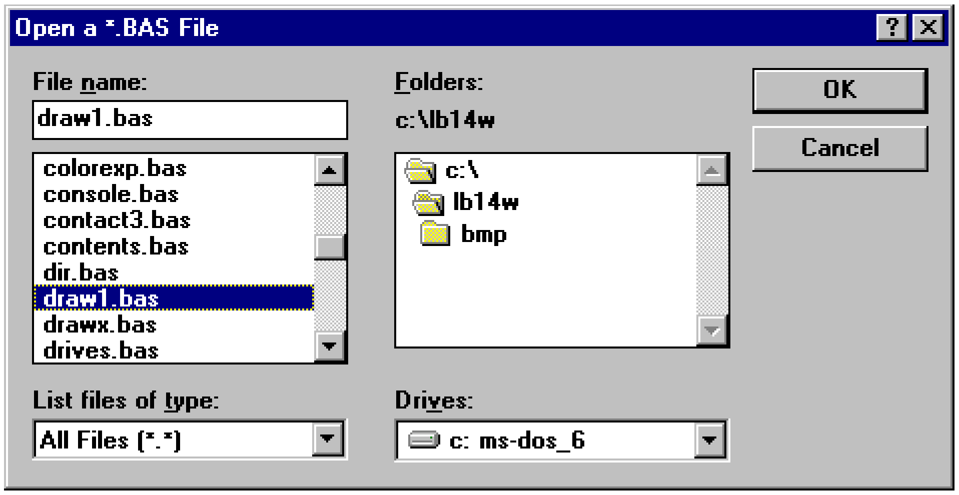
This is where code is written, and this is where you will spend most of your time when writing Liberty BASIC programs. Notice the various pull-down menus along the top of the window. These are for loading and saving files, editing, running/debugging, setting up configuration, and getting help.

Running an example program:

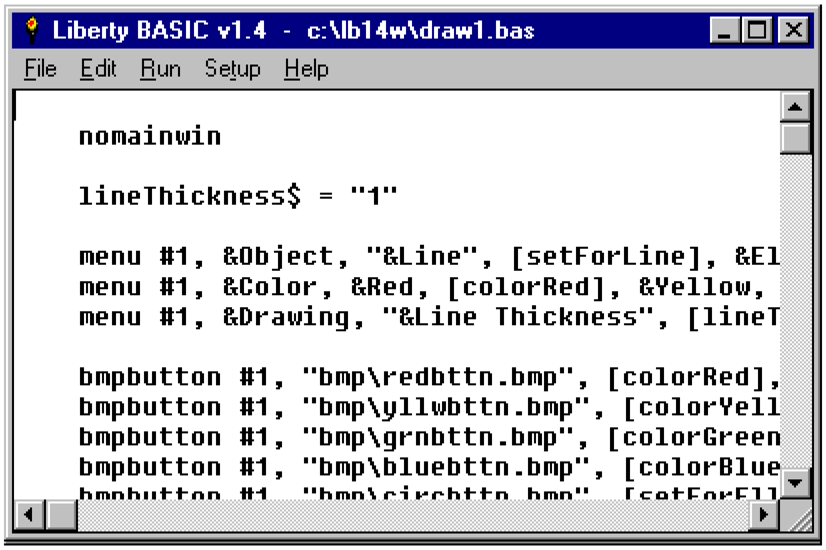
Let's begin our exploration of Liberty BASIC by running one of the sample programs we've provided. Pull down the File menu and select the Open item as shown.



You will see a dialog box similar to the one displayed below. Inside the box titled Files: there is a list of files that you can load. These are text files containing our example BASIC programs. Select the item named draw1.bas and click on OK.

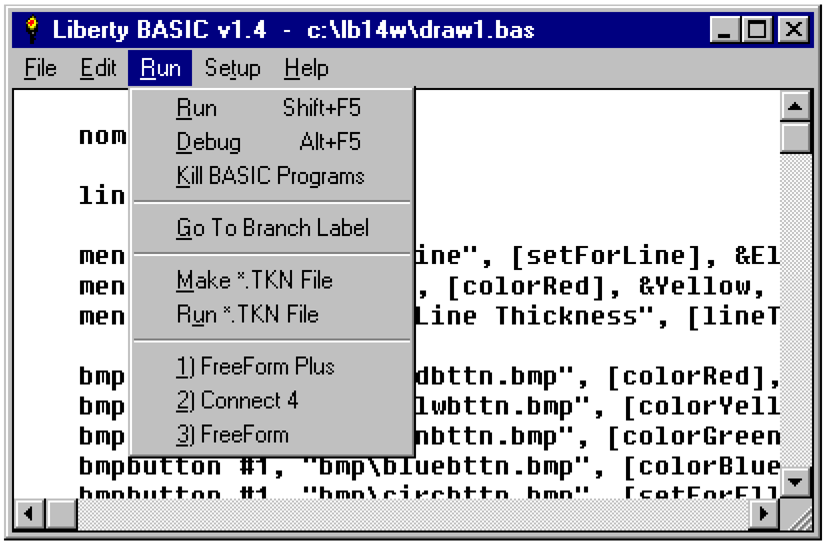


Liberty BASIC will load the draw1.bas program you selected. The result will look like this window. This is BASIC code for a Windows drawing program. As you learn to program in Liberty BASIC you will be able to extend this program and the other included samples to do what you want. But right now let's see how it runs!



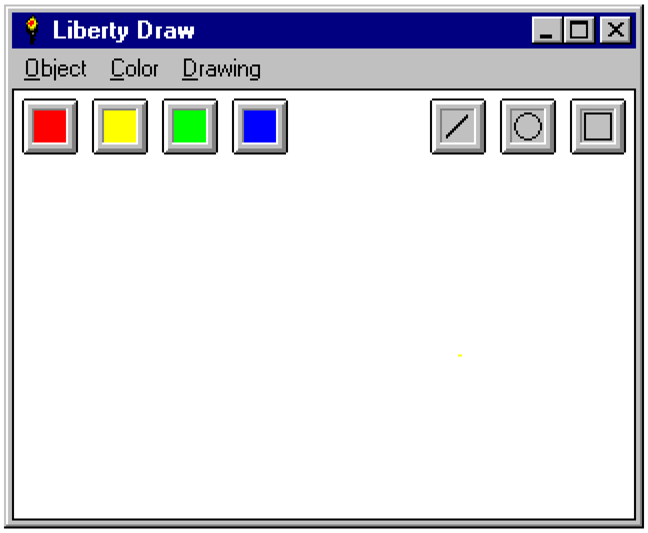
```
    nomainwin

    lineThickness$ = "1"

    menu #1, &Object, "&Line", [setForLine], &El
    menu #1, &Color, &Red, [colorRed], &Yellow,
    menu #1, &Drawing, "&Line Thickness", [lineT

    bmpbutton #1, "bmp\redbttn.bmp", [colorRed],
    bmpbutton #1, "bmp\yllwbttn.bmp", [colorYell
    bmpbutton #1, "bmp\grnbttn.bmp", [colorGreen
    bmpbutton #1, "bmp\bluebttn.bmp", [colorBlue
    bmpbutton #1, "bmp\circbttn.bmp", [setForEll
```

Running a Liberty BASIC program is easy. Select the Run menu and mouse click on the Run item, as illustrated below.

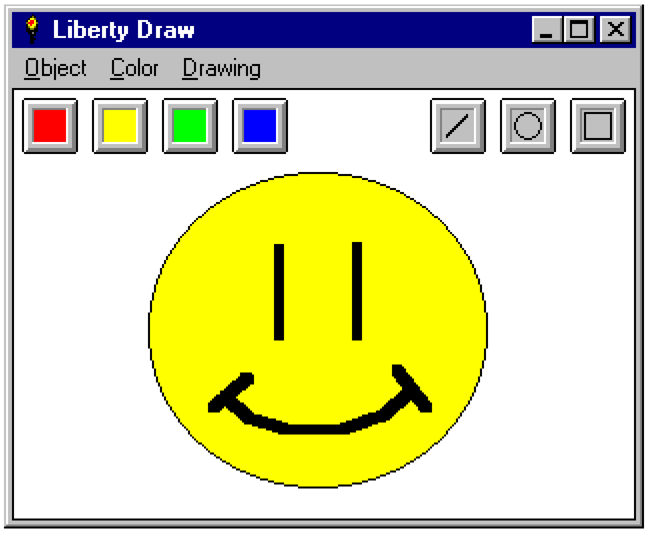


```
    nom

    lin

    men                    ine", [setForLine], &El
    men                    , [colorRed], &Yellow,
    men                    Line Thickness", [lineT

    bmp                    dbttn.bmp", [colorRed],
    bmp                    lwbttn.bmp", [colorYell
    bmp                    nbttn.bmp", [colorGreen
    bmpbutton #1, "bmp\bluebttn.bmp", [colorBlue
    bmpbutton #1, "bmp\circbttn.bmp", [setForEll
```

Now Liberty BASIC will take a few seconds to compile and run our drawing program (some computers will take longer than others). When it is finished compiling, a window belonging to our drawing program will appear:



Let's try drawing a little something with Liberty Draw!



Feel free to play with Liberty Draw, and when you're done close its window.

Command Reference

Here is an alphabetical list of the Liberty BASIC commands and functions:

more (skip to next page)

ABS(n)	absolute value
ACS(n)	arc-cosine of n
ASC(s$)	ascii value of s$
ASN(n)	arc-sine of n
ATN(n)	arc-tangent of n
BEEP	ring bell
BMPBUTTON	add a bitmap button to a window
BMPSAVE	save a bitmap from memory to a *.bmp file
BUTTON	add a button to a window
CALLDLL	call an API or DLL function
CHECKBOX	add a checkbox to a window
CHR$(n)	return character of ascii value n
CLOSE #h	close a file or window with handle #h
CLS	clear a program's mainwindow
CommandLine$	contains any command line switches used on startup
CONFIRM	opens a confirm dialog box
COS(n)	cosine of n
CURSOR	changes the mouse cursor
DATE$( )	returns string with today's date
DefaultDir$	a variable containing the default directory
DIM array( )	dimension array( )
DisplayWidth	a variable containing the width of the display
DisplayHeight	a variable containing the height of the display
Drive$	special variable, holds drive letters
DUMP	force the LPRINT buffer to print
EOF(#h)	end-of-file status for #h
END	marks end of program execution
EXP(n)	returns e^n logarithm
FIELD #h, list...	sets random access fields for #h
FILEDIALOG	opens a file selection dialog box
FILES	returns file and subdirectory info
FOR...NEXT	performs looping action
GET #h, n	get random access record n for #h
GETTRIM #h, n	get r/a record n for #h, blanks trimmed
GOSUB label	call subroutine label
GOTO label	branch to label
GRAPHICSBOX	add a graphics region to a window
HEXDEC("value")	convert a hexadecimal string to a decimal value
HWND(#handle)	return a Windows handle for a window
IF...THEN	perform conditional action(s)
IF THEN ELSE	perform conditional action(s)
INP(port)	get a byte value from an I/O port
INPUT	get data from keyboard, file or button
INPUT$(#h, n)	get n chars from handle #h
INSTR(a$,b$,n)	search for b$ in a$, with optional start n
INT(n)	integer portion of n
KILL s$	delete file named s$
LEFT$(s$, n)	first n characters of s$

LEN(s$)	length of s$
LET var = expr	assign value of expr to var
more...	

Command Reference - Part 2

Back to Command Reference - Part 1

LINE INPUT	get next line of text from file
LOADBMP	load a bitmap into memory
LOF(#h)	return length of open file #h
LOG(n)	natural log of n
LOWER$(s$)	s$ converted to all lowercase
LPRINT	print to hard copy
MENU	adds a pull-down menu to a window
MID$( )	return a substring from a string
MKDIR( )	make a new subdirectory
NAME a$ AS b$	rename file named a$ to b$
NOMAINWIN	keep a program's main window from opening
NOTICE	open a notice dialog box
OPEN	open a file or window
OPEN "COMn:..."	open a communications port for reading/writing
OUT port, byte	send a byte to a port
Platform$	special variable containing platform name
PLAYWAVE	plays a *.wav sound file
PRINT	print to a file or window
PROMPT	open a prompter dialog box
PUT #h, n	puts a random access record n for #h
RADIOBUTTON	adds a radiobutton to a window
REDIM	redimensions an array and resets it contents
REM	adds a remark to a program
RETURN	return from a subroutine call
RIGHT$(s$, n)	n rightmost characters of s$
RMDIR( )	remove a subdirectory
RND(n)	return pseudo-random seed
RUN s$, mode	run external program s$, with optional mode
SCAN	checks for and dispatches user actions
SIN(n)	sine of n
SORT	sorts single and double dim'd arrays
SPACE$(n)	returns a string of n spaces
STR$(n)	returns string equivalent of n
STRUCT	builds a structure used in calling of APIs and DLL functions
TAN(n)	tangent of n
TIME$( )	returns current time as string
TRACE n	sets debug trace level to n
TRIM$(s$)	returns s$ without leading/trailing spaces
UPPER$(s$)	s$ converted to all uppercase
USING( )	performs numeric formatting
UpperLeftX	specifies the x part of the position where the next window will open
UpperLeftY	specifies the y part of the position where the next window will open
VAL(s$)	returns numeric equivalent of s$
Version$	special variable containing LB version info
WHILE...WEND	performs looping action
WindowWidth	specifies the width of the next window to open
WindowHeight	specifies the height of the next window to open
WORD$(s$, n)	returns nth word from s$

ABS(n)

Description:

This function returns | n | (the absolute value of n).

Usage:

print abs(-5)	produces: 5
print abs(6 - 13)	produces: 7

ACS(n)

Description:

Returns the arc-cosine of the number n.

Usage:

```
for c = 0 to 1 step 0.01
  print "The arc-cosine of "; c; " is "; acs(c)
next c
```

Note: See also COS( )

ASC(s$)

Description:

This function returns the ASCII value of the first character of string s$.

Usage:

```
print asc( "A" )              produces:  65

let name$ = "Tim"
firstLetter = asc(name$)
print firstLetter             produces:  84

print asc( "" )               produces:  0
```

ASN(n)

Description:

Returns the arcsine of the number n.

Usage:

```
for c = 0 to 1 step 0.01
  print "The arcsine of "; c; " is "; asn(c)
next c
```

Note: See also SIN( )

ATN(n)

Description:

Returns the arc-tangent of the number n.

Usage:

```
for c = 1 to 45
  print "The arctangent of "; c; " is "; atn(c)
next c
```

Note: See also TAN( )

BEEP

Description:

Returns the arc-tangent of the number n.

Usage:

```
for c = 1 to 45
  print "The arctangent of "; c; " is "; atn(c)
next c
```

Note: See also TAN( )

BMPBUTTON

BMPBUTTON #handle.ext, filespec, return, corner, posx, posy

Description:

This statement lets you add bitmapped buttons to windows that you open. The main program window cannot have buttons added, but any window that you create via the OPEN command can have as many buttons as you want.

Usage:

Before you actually OPEN the window, each bitmapped button must be declared with a BMPBUTTON statement. Here is a brief description for each parameter as listed above:

#handle.ext - You must use the same handle that will be used for the window that the button will belong to, plus an optional unique extension.

filespec - The full pathname of the *.bmp file containing the bitmap for the button you are creating. The button will be the same size as the bitmap.

return - Again, use only one word and do not bound it with quotes or use a string variable. If return is set to a valid branch label, then when the button is pressed, execution will restart there (just as with GOTO or GOSUB), but if return is not a valid branch label, then the value of return is used as input to a specified variable (as in input a$).

corner - UL, UR, LL, or LR specifies which corner of the window to anchor the button to. For example, if LR is used, then the button will appear in the lower right corner. UL = upper left, UR = upper right, LL = lower left, and LR = lower right

posx, posy - These parameters determine how to place the button relative to the corner it has been anchored to. For example if corner is LR, posx is 5, and posy is 5, then the button will be 5 pixels up and left of the lower right corner. Another way to use posx & posy is to use values less than one. For example, if corner is UL, posx is .9, and posy is .9, then the button will be positioned 9/10th of the distance of the window in both x and y from the upper left corner (and thus appear to be anchored to the lower right corner).

A collection of button *.bmp has been included with Liberty BASIC, including blanks. Windows Paint can be used to edit and make buttons for Liberty BASIC.

Program execution must be halted at an input statement in order for a button press to be read and acted upon.

See also: BUTTON, MENU

BUTTON

BUTTON #handle.ext, label, return, corner, posx, posy

Description:

This statement lets you add buttons to windows that you open.   The main program window cannot have buttons added, but any window that you create via the OPEN command can have as many buttons as you want.

Usage:

Before you actually OPEN the window, each button must be declared with a BUTTON statement.   Here is a brief description for each parameter as listed above:

#handle.ext - You must use the same handle that will be used for the window that the button will belong to, plus an optional unique extension.

label - Type the label desired for the button here.   Do not bound the word with quotes, and do not use a string variable.

return - Again, use only one word and do not bound it with quotes or use a string variable.   If return is set to a valid branch label, then when the button is pressed, execution will restart there (just as with GOTO or GOSUB), but if return is not a valid branch label, then the value of return is used as input to a specified variable (as in input a$).

corner - UL, UR, LL, or LR specifies which corner of the window to anchor the button to.   For example, if LR is used, then the button will appear in the lower right corner.   UL = upper left, UR = upper right, LL = lower left, and LR = lower right

posx, posy - These parameters determine how to place the button relative to the corner it has been anchored to.   For example if corner is LR, posx is 5, and posy is 5, then the button will be 5 pixels up and left of the lower right corner.   Another way to use posx & posy is to use values less than one.   For example, if corner is UL, posx is .9, and posy is .9, then the button will be positioned 9/10th of the distance of the window in both x and y from the upper left corner (and thus appear to be anchored to the lower right corner).

Program execution must be halted at an input statement in order for a button press to be read and acted upon.   See next page.

Here is a sample program:

```
   ' this button will be labeled Sample and will be located
   ' in the lower right corner.  When it is pressed, program
   ' execution will transfer to [test]

  button #graph, Bell, [bell], LR, 5, 5

   ' this button will be labeled Example and will be located
   ' in the lower left corner.  When it is pressed, the string
   ' "Example" will be returned.

   button #graph, Quit, [quit], LL, 5, 5

   ' open a window for graphics
     open "Button Sample" for graphics as #graph

   ' print a message in the window
     print #graph, "\\This is a test"
     print #graph, "flush"

   ' get button input
 [loop]
   input b$   ' stop and wait for a button to be pressed
   if b$ = "Example" then [example]
   goto [loop]

  ' the Sample button has been pressed, ring the terminal bell
  ' and close the window
 [bell]
   beep
   close #graph
   end

  ' The Example button has been pressed, close the window
  ' without ringing the bell
 [quit]
   close #graph
   end
```

CALLDLL

CALLDLL #handle, "function", parm1 as type1 [, parm2 as type2], return as returnType

Description:

CALLDLL is used to call functions from the Windows API or from a third party DLL (or one you might write yourself).

Usage:

Here is an example of calling an API that minimizes a window and changes its label text.

```
open "An Example" for window as #main
h = hwnd(#main)

open "user" for dll as #user

calldll #user, "CloseWindow", _
    h as word, _
    result as void

calldll #user, "SetWindowText", _
    h as word, _
    "I was minimized!" as ptr, _
    result as void

close #user
```

Here is an example of CALLDLL that uses a struct built using the STRUCT statement.   This demonstrates how to get the corner positions of a window.

```
struct winRect, _
    orgX as uShort, _
    orgY as uShort, _
    cornerX as uShort, _
    cornerY as uShort

open "An Example" for window as #main
hMain = hwnd(#main)

open "user.dll" for dll as #user
calldll #user, "GetWindowRect", _
    hMain as word, _
    winRect as struct, _
    result as void

close #user
close #main

print "Window position upperLeft: "; _
    winRect.orgX.struct; ", "; winRect.orgY.struct

print "Window position lowerRight: "; _
```

```
        winRect.cornerX.struct; " x "; winRect.cornerY.struct
```

Using Types

The CALLDLL statement requires that each parameter passed be typed. Simple data types in Windows programming are often just renamed versions of the types below, so just substitute the correct ones.

Here they are:

double	(a double float)
dword, ulong	(4 bytes)
handle	(2 bytes)
long	(4 bytes)
short	(2 bytes)
ushort, word	(2 bytes, substitute this for boolean)
ptr	(4 bytes, long pointer, for passing strings)
struct	(4 bytes, long pointer, for passing structs)
void, none	(a return type only)

Notice: If you know you will be passing a negative number as a parameter, first convert the number to a unsigned integer value. This can be done in Liberty BASIC using the OR operator like so:

```
    variableName  =  -1 or 0    'convert -1 to an unsigned value
```

CHECKBOX

CHECKBOX #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height

Description:

Adds a checkbox control to the window referenced by #handle. Checkboxes have two states, set and reset. They are useful for getting input of on/off type information.

Here is a description of the parameters of the CHECKBOX statement:

"label"	- This contains the visible text of the checkbox
[set]	- This is the branch label to goto when the user sets the checkbox by clicking on it.
[reset]	- This is the branch label to goto when the user resets the checkbox by clicking on it.
xOrigin	- This is the x position of the checkbox relative to the upper left corner of the window it belongs to.
yOrigin	- This is the y position of the checkbox relative to the upper left corner of the window it belongs to.
width	- This is the width of the checkbox control
height	- This is the height of the checkbox control

Usage: See the included program checkbox.bas for an example of how to use checkboxes

CHR$(n)

Description:

Returns a one character long string, consisting of the character represented on the ASCII table by the value n (0 - 255).

Usage:

```
' print each seperate word in text$ on its own line
text$ = "now is the time for all great men to rise"
for index = 1 to len(text$)
    c$ = mid$(text$, index, 1)
    ' if c$ is a space, change it to a carraige return
    if c$ = chr$(32) then c$ = chr$(13)
    print c$ ;
next index
```

Produces:

```
now
is
the
time
for
all
great
men
to
rise
```

CLOSE #h

Description:

This command is used to close files and devices. This is the last step of a file read and/or write, or to close graphic, spreadsheet, or other windows when finished with them. If when execution of a program is complete there are any files or devices left open, Liberty BASIC will display a dialog informing you that it found it necessary to close the opened files or devices. This is designed as an aid for you so that you will be able to correct the problem. If on the other hand you choose to terminate the program early (this is done by closing the program's main window before the program finishes), then Liberty BASIC will close any open files or devices without posting a notice to that effect.

Usage:

```
  open "Graphic" for graphics as #gWin  ' open a graphics window
  print #gWin, "home"                   ' center the pen
  print #gWin, "down"                   ' put the pen down
  for index = 1 to 100                  ' loop 100 times
    print #gWin, "go "; index           ' move the pen foreward
    print #gWin, "turn 63"              ' turn 63 degrees
  next index
  input "Press 'Return'."; r$           ' this appears in main window
  close #gWin                           ' close graphic window
```

CLS

Description:

Clears the main program window of text and sets the cursor back at the upper left hand corner.   Useful for providing a break to seperate different sections of a program functionally.   Additionally, since the main window doesn't actually discard past information on its own, the CLS command can be used to reclaim memory from your program by forcing the main window to dump old text.

Usage:

```
  print "The total is: "; grandTotal
  input "Press 'Return' to continue."; r$
  cls
  print "*** Enter Next Round of Figures ***"
```

CONFIRM

CONFIRM

CONFIRM string; responseVar

Description:

This statement opens a dialog box displaying the contents of string and presenting two buttons marked 'Yes' and 'No'. When the selection is made, the string "yes" is returned if 'Yes' is pressed, and the string "no" is returned if 'No' is pressed. The result is placed in responseVar.

Usage:

```
[quit]

  ' bring up a confirmation box to be sure that
  ' the user wants to quit
  confirm "Are you sure you want to QUIT?"; answer$
  if answer$ = "no" then [mainLoop]
  end
```

COS(n)

COS(n)

Description:

Returns the cosine of the number n.

Usage:

```
for c = 1 to 45
  print "The cosine of "; c; " is "; cos(c)
next c
```

Note: See also SIN( ) and TAN( )

DATE$( )

Description:

Instead of adopting MBASIC's date$ variable, we decided to use a function instead, figuring that this might give us additional flexibility later.   This function returns the current date in long format.

Usage:

```
  print date$( )
```

Produces:

```
  Feb 5, 1991
```

Or you can assign a variable the result:

```
  d$ = date$( )
```

DefaultDir$

Description:

A string variable that contains the default directory for the running Liberty BASIC program.   The format is "drive:\dir1", or "drive:\dir1\dir2" etc.

Usage:

```
    print DefaultDir$
```

Or you might use it with a FILES statement:

```
    files DefaultDir$, "*.txt", info$(
```

DIM array( )

DIM array(size)   -or-   DIM array(size, size)

Description:

DIM sets the maximum size of an array. Any array can be dimensioned to have as many elements as memory allows. If an array is not DIMensioned explicitly, then the array will be limited to 10 elements, 0 to 9. Non DIMensioned double subscript arrays will be limited to 100 elements 0 to 9 by 0 to 9.

Usage:

```
print "Please enter 10 names."
for index = 0 to 9
  input names$ : name$(index) = name$
next index
```

The FOR . . . NEXT loop in this example is limited to a maximum value of 9 because the array names$ ( ) is not dimensioned, and therefore is limited to 10 elements. To remedy this problem, we can add a DIM statement, like so:

```
dim names$(20)
print "Please enter 20 names."
for index = 0 to 19
  input names$ : names$(index) = name$
next index
```

Double subscripted arrays can store information more flexibly, like so:

```
dim customerInfo$(10, 5)
print "Please enter information for 10 customers."
for index = 0 to 9
  input "Customer name >"; info$ : customerInfo$(index, 0) = info$
  input "Address >"; info$ : customerInfo$(index, 1) = info$
  input "City >"; info$ : customerInfo$(index, 2) = info$
  input "State >"; info$ : customerInfo$(index, 3) = info$
  input "Zip >"; info$ : customerInfo$(index, 4) = info$
next index
```

Drive$

Description:

Drives$ is a system variable.   You can operate on it like any other variable.   Use it in expressions, print it, perform functions on it, etc.   It's special in that it contains the drive letters for all the drives installed on in the computer in use.

For example:

```
    print Drives$
```

Would in many cases produce:

```
    a: b: c:
```

Or you could use it to provide a way to select a drive like this:

```
    'a simple example illustrating the use of the Drives$ variable
    dim letters$(25)
    index = 0
    while word$(Drives$, index + 1) <> ""
        letters$(index) = word$(Drives$, index + 1)
        index = index + 1
    wend

    statictext #select, "Double-click to pick a drive:", 10, 10, 200, 20
    listbox #select.list, letters$(, [selectionMade], 10, 35, 100, 150
    open "Scan drive" for dialog as #select

    input r$

[selectionMade]

    close #select
    end
```

DUMP

Description:

Forces anything that has been LPRINTed to be sent to the Print Manager.

Usage:

```
'sample program using LPRINT and DUMP
open "c:\autoexec.bat" for input as #source
while eof( #source ) = 0
  line input #source, text$        'print each line
  lprint text$
wend
close #source
dump         'force the print job
end
```

Note: see also LPRINT

EOF(#h)

Description:

Used to determine when reading from a sequential file whether the end of the file has been reached. If so, -1 is returned, otherwise 0 is returned.

Usage:

```
   open "testfile" for input as #1
   if eof(#1) < 0 then [skipIt]
[loop]
   input #1, text$
   print text$
   if eof(#1) = 0 then [loop]
[skipIt]
   close #1
```

END

Description:

Used to immediately terminate execution of a program.   If any files or devices are still open (see CLOSE) when execution is terminated, then Liberty BASIC will close them for you and present you with a dialog expressing this fact.   It is good programming practice to close files and devices before terminating execution.

   Note: The STOP statement is functionally identical to END and is
   interchangable

Usage:

```
   .
   .
   print "Preliminary Tests Complete."
[askAgain]
   input "Would you like to continue (Y/N) ?"; yesOrNo$
   yesOrNo$ = left$(yesOrNo$, 1)
   if yesOrNo$ = "y" or yesOrNo$ = "Y" then [continueA]
   ifYesOrNo$ = 'n" or yesOrNo$ = "N" then end
   print "Please answer Y or N."
   goto [askAgain]
[continueA]
```

EXP(n)

Description:

This function returns e ^ n,   e being 2.7182818 . . .

Usage:

```
print exp( 5 )                    produces:   148.41315
```

FIELD #h, list...

FIELD #handle, # as varName, # as varName, . . .

Description:

FIELD is used with an OPEN "filename.ext" for random as #handle statement to specify the fields of data in each record of the opened file.   For example in this program FIELD sets up 6 fields of data, each with an appropriate length, and associates each with a string variable that holds the data to be stored in that field:

```
  open "custdata.001" for random as #cust len = 70     ' open as random access
  field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

[inputLoop]
  input "Name >"; name$
  input "Street >"; street$
  input "City >"; city$
  input "State >"; state$
  input "Zip Code >"; zip$
  input "Age >"; age

  confirm "Is this entry correct?"; yesNo$     ' ask if the data is entered correctly
  if   yesNo$ = "no" then [inputLoop]

  recNumber = recNumber + 1     ' add 1 to the record # and put the record
  put #cust, recNumber

  confirm "Enter more records?"; yesNo$     ' ask whether to enter more records
  if yesNo$ = "yes" then [inputLoop]

  close #cust     ' end of program, close file
  end
```

Notice that Liberty BASIC permits the use of numeric variables in FIELD (eg. age), and it allows you to PUT and GET with both string and numeric variables, automatically, without needing LSET, RSET, MKI$, MKS$, MKD$, CVI, CVS, & CVD that are required with Microsoft BASICs.

Note: See also PUT and GET

FILEDIALOG

FILEDIALOG titleString, templateString, receiverVar$

Description:

This command opens a file dialog box. The titleString is used to label the dialog box. The templateString is used as a filter to list only files matching a wildcard, or to place a full suggested filename.

The box lets you navigate around the directory structure, looking at files that have a specific extension. You can then select one, and the resulting full path specification will be placed into receiverVar$, above.

The following example would produce a dialog box asking the user to select a text file to open:

```
    filedialog "Open text file", "*.txt", fileName$
```

If then summary.txt were selected, and OK clicked, then program execution would resume after placing the string "c:\liberty\summary.txt" into fileName$.

If on the other hand Cancel were clicked, then an empty string would be placed into fileName$. Program execution would then resume.

Look at the program grapher1.bas for a practical application of this command.

FILES

Description:

The FILES statement collects file and directory information from any disk and or directory and fills a double-dimensioned array with the information. Also good for determining if a specific file exists (see below).

Usage:

```
'you must predimension info$(, even though FILES will
'redimension it to fit the information it provides.
dim info$(10, 10)
files "c:\", info$(
```

The above FILES statement will fill info$( ) in this fashion:

```
info$(0, 0) - a string specifying the qty of files found
info$(0, 1) - a string specifying the qty of subdirectories found
info$(0, 2) - the drive spec
info$(0, 3) -   the directory path
```

Starting at info$(1, x) you will have file information like so:

```
info$(1, 0) - the file name
info$(1, 1) - the file size
info$(1, 2) - the file date/time stamp
```

Knowing from info$(0, 0) how many files we have (call it n), we know that our subdirectory information starts at n + 1, so:

```
info$(n + 1, 0) - the complete path of a directory entry (ie. \work\math)
info$(n + 1, 1) - the name of the directory in specified (ie. math)
```

You can optionally specify a wildcard. This lets you get a list of all *.ini files, for example. This is how you do it:

```
files DefaultDir$, "*.ini", info$(
```

This also makes it practical to use to check for file existance. If you want to know if a file c:\config.bak exists, you could try...

```
files "c:\", "config.bak", info$(
```

If val(info$(0, 0)) > 0, then the file exists.

See the dir.bas example included.

FOR...NEXT

FOR...NEXT

Description:

The FOR . . . NEXT looping construct provides a way to execute code a specific amount of times.   A starting and ending value are specified like so:

```
    for var = 1 to 10
      {BASIC code}
    next var
```

In this case, the {BASIC code} is executed 10 times, with var being 1 the first time, 2 the second, and on through 10 the tenth time.   Optionally (and usually) var is used in some calculation(s) in the {BASIC code}. For example if the {BASIC code} is   print var ^ 2, then a list of squares for var will be displayed upon execution.

The specified range could just as easily be 2 TO 20, instead of 1 TO 10, but since the loop always counts +1 at a time, the first number must be less than the second.   The way around this limitation is to place STEP n at the end of for FOR statement like so:

```
    for index = 20 to 2 step -1
      {BASIC code}
    next index
```

This would loop 19 times returning values for index that start with 20 and end with 2.   STEP can be used with both positive and and negative numbers and it is not limited to integer values.   For example:

```
    for x = 0 to 1 step .01
      print "The sine of "; x; " is "; sin(x)
    next x
```

NOTE: It is not recommended to pass control of a program out of a   FOR   . . . NEXT loop using GOTO (GOSUB is acceptable).   Liberty BASIC may behave unpredictably.   For example:

```
  for index = 1 to 10
    print "Enter Customer # "; index
    input customer$
    if customer$ = "" then [quitEntry]    ' <- don't cut out of a for ... next loop like this
    cust$(index) = customer$
  next index
  [quitEntry]
```

. . . is not allowed!   Rather use while ... wend:

```
  index = 1
  while customer$ <> "" and index <= 10
    print "Enter Customer # "; index
    input customer$
    cust$(index) = customer$
    index = index + 1
  wend
```

GET #h, n

GET #handle, recordNumber

Description:

GET is used after a random access file is opened to get a record of information (see FIELD) out of the file from a specified position.

Usage:

```
  open "custdata.001" for random as #cust len = 70     ' open random access file
  field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

  ' get the data from record 1
  get #cust, 1

  print name$
  print street$
  print city$
  print state$
  print zip$
  print age

  close #cust
  end
```

Note: See also PUT, FIELD

GETTRIM #h, n

GETTRIM #handle, recordNumber

Description:

The GETTRIM command is exactly like the GET command, but when data is retrieved, all leading and trailing blank space is removed from all data fields before being committed to variables.

Note: see also GET

GOSUB label

Description:

GOSUB causes execution to proceed to the program code following the label if it exists,   using the form 'GOSUB label'.   The label can be either a traditional line number or a branch label in the format [???????] where the ?'s can be any upper/lowercase letter combination.   Spaces and numbers are not allowed.

Here are some valid branch labels: [mainMenu]   [enterLimits]   [repeatHere]
Here are some invalid branch labels: [enter limits]   mainMenu   [1moreTime]

After execution is transferred to the point of the branch label, then each statement will be executed in normal fashion until a RETURN is encountered. When this happens, execution is transferred back to the statement immediately after the GOSUB.   The section of code between a GOSUB and its RETURN is known as a 'subroutine.'   One purpose of a subroutine is to save memory by having only one copy of code that is used many times throughout a program.

Usage:

```
  .
  .
  print "Do you want to continue?"
  gosub [yesOrNo]
  if answer$ = "N" then [quit]
  print "Would you like to repeat the last sequence?"
  gosub [yesOrNo]
  if answer$ = "Y" then [repeat]
  goto [generateNew]

[yesOrNo]
  input answer$
  answer$ = left$(answer$, 1)
  if answer$ = "y" then answer$ = "Y"
  if answer$ = "n" then answer$ = "N"
  if answer$ = "Y" or answer$ = "N" then return
  print "Please answer Y or N."
  goto [yesOrNo]
  .
  .
```

You can see how using GOSUB [yesOrNo] in this case saves many lines of code in this example.   The subroutine [yesOrNo] could easily be used many other times in such a hypothetical program, saving memory and reducing typing time and effort.   This reduces errors and increases productivity.

Note: see also GOTO

GOTO label

Description:

GOTO causes Liberty BASIC to proceed to the program code following the label if one exists, using the form 'GOTO label'. The label can be either a traditional line number or a branch label in the format [???????] where the ?'s can be any upper/lowercase letter combination. Spaces and digits are not allowed.

Here are some valid branch labels: [mainMenu] [enterLimits] [repeatHere]
Here are some invalid branch labels: [enter limits] mainMenu [1moreTime]

Usage:

```
  .
  .
[repeat]
  .
  .
[askAgain]
  print "Make your selection (m, r, x)."
  input selection$
  if selection$ = "M" then goto [menu]
  if selection$ = "R" then goto [repeat]
  if selection$ = "X" then goto [exit]
  goto [askAgain]
  .
  .
[menu]
  print "Here is the main menu."
  .
  .
[exit]
  print "Okay, bye."
  end
```

Notes:

In the lines containing IF . . . THEN GOTO, the GOTO is optional.

The expression IF . . . THEN [menu] is just as valid as
IF . . . THEN GOTO [menu]. But in the line GOTO [askAgain], the GOTO
is required.

See also GOSUB

GRAPHICSBOX

GRAPHICSBOX #handle.ext, xOrigin, yOrigin, width, height

Description:

Adds a region in a window that can perform graphics operations.

Usage:

```
'open a window with a graphicsbox
graphicsbox #main.graph, 10, 10, 100, 100
open "Graphics" for window as #main
```

The above graphicsbox responds to all the commands that a graphics window does (except trapclose). See the GUI Programming help file for more information.

See the list of graphics commands

HWND(#handle)

Description:

Returns the value of the Windows handle for the window referred to by the Liberty BASIC #handle. This is very useful for making many different kinds of Windows API calls, and many third-party DLL will require that this value be provided for their own purposes.

Usage:

```
    open "Example" for window as #1
    h1 = hwnd(#1)
```

IF...THEN

IF expression THEN expression(s)

Description:

The purpose of IF . . . THEN is to provide a mechanism for your computer software to make decisions based on the data available. A decision-making mechanism is used in very simple situations and can be used in combinations to engineer solutions to problems of great complexity.

The expression (see above) is a boolean expression (meaning that it evaluates to a true or false condition). In this expression we place the logic of our decision-making process. For example, if we are writing a inventory application, and we need to know when any item drops below a certain level in inventory, then our decision-making logic might look like this:

```
.
.
if level <= reorderLevel then expression(s)
next BASIC program line
.
.
```

The 'level <= reorderLevel' part of the above expression will evaluate to either true or false. If the result was true, then the expression(s) part of that line (consisting of a branch label or any valid BASIC statements) will be executed. Otherwise execution will immediately begin at the next BASIC program line.

The following are permitted:

```
if a < b then [lessThan]
```

This causes program execution to begin at branch label [lessThan] if a is less than b.

```
if sample < lowLimit or sample > highLimit then beep : print"Out of range!"
```

This causes the terminal bell to ring and the message Out of range! to be displayed if sample is less than lowLimit or greater then highLimit.

Note: see also IF...THEN...ELSE

IF THEN ELSE

IF expression THEN expression(s)1 ELSE expression(s)2

Description:

This extended form of IF . . . THEN adds expressiveness and simplifies coding of some logical decision-making software. Here is an example of its usefulness.

Consider:

```
[retry]
   input"Please choose mode, (N)ovice or e(X)pert?"; mode$
   if len(mode$) = 0 then print "Invalid entry! Retry" : goto [retry]
   mode$ = left$(mode$, 1)
   if instr("NnXx", mode$) = 0 then print "Invalid entry! Retry" : goto [retry]
   if instr("Nn", mode$) > 0 then print "Novice mode" : goto [main]
   print "eXpert mode"
[main]
   print "Main Selection Menu"
```

Look at the two lines before the [main] branch label. The first of these two lines is required to branch over the next line. These lines can be shortened to one line as follows:

```
   if instr("Nn",mode$)> 0 then print "Novice mode" else print "eXpert mode"
```

Some permitted forms are as follows:

```
   if a < b then statement else statement
   if a < b then [label] else statement
   if a < b then statement else [label]
   if a < b then statement : statement else statement
   if a < b then statement else statement : statement
   if a < b then statement : goto [label] else statement
   if a < b then gosub [label1] else gosub [label2]
```

Any number of variations on these formats are permissible. The a < b boolean expression is of course only a simple example chosen for convenience.

You must replace it with the correct expression to suit your problem.

Note: see also IF...THEN

INPUT

INPUT #handle "string expression"; variableName

Description:

This command has several possible forms:

```
  input var

      - stop and wait for user to enter data in the program's
      main window and press the 'Return' key, then assign
      the data entered to var.

  input "enter data"; var

      - display the string "enter data" and then stop
      and wait for user to enter data in the program's main window
      and press 'Return', then assign the data entered to var.

  input #name, var

      - Get the next data item from the open file or device using handle
      named #handle and assign the data to var.   If no device or file exists
      that uses the handle named #handle, then return an error.


  input #name, var1, var2

      - The next two data items are fetched and assigned to var1 and var2.


  line input #name, var$

       - The line input statement will read from the file, ignoring commas in the
      input stream and completing the data item only at the next carraige
      return or at the end of file.   This is useful for reading text with embedded
      commas
```

Usage:

```
  'Display a text file
  input "Please type a filename >";   filename$
  open filename$ for input as #text
[loop]
  if eof(#text) <> 0 then [quit]
  input #text, item$
  print item$
  goto [loop]
[quit]
  close #text
  print "Done."
  end
```

Note: In Liberty BASIC, INPUT cannot be used to input data directly into arrays, only into the simpler variables.

```
   input a$(1)                         - is illegal
   input string$ : a$(1) = string$          - use this instead
```

Most versions of Microsoft BASIC implement INPUT to automatically place a question mark on the display in front of the cursor when the user is prompted for information like so:

```
   input "Please enter the upper limit"; limit

   produces:

      Please enter the upper limit ? |
```

Liberty BASIC permits you the luxury of deciding for yourself whether the question mark appears at all.

```
   input "Please enter the upper limit :"; limit

   produces:

      Please enter the upper limit: |

   and:      input limit      produces simply:

      ? |
```

In the simple form input limit, the question mark is inserted automatically, but if you do specify a prompt, as in the above example, only the contents of the prompt are displayed, and nothing more. If for some reason you wish to input without a prompt and without a question mark, then the following will achieve the desired effect:

```
   input ""; limit
```

Additionally, in most Microsoft BASICs, if INPUT expects a numeric value and a non numeric or string value is entered, the user will be faced with a comment something like 'Redo From Start', and be expected to reenter. Liberty BASIC does not automatically do this, but converts the entry to a zero value and sets the variable accordingly. This is not considered a problem but rather a language feature, allowing you to decide for yourself how your program will respond to the situation.

One last note: In Liberty BASIC input prompt$; limit is also valid. Try:

```
   prompt$ = "Please enter the upper limit:"
   input prompt$; limit
```

INPUT$(#h, n)

INPUT$(#handle, items)

Description:

Permits the retrieval of a specified number of items from an open file or device using #handle. If #handle does not refer to an open file or device then an error will be reported.

Usage:

```
   'read and display a file one character at a time
   open "c:\autoexec.bat" for input as #1
[loop]
      if eof(#1) <> 0 then [quit]
      print input$(#1, 1);
      goto [loop]
[quit]
      close #1
      end
```

For most devices (unlike disk files), one item does not refer a single character, but INPUT$( ) may return items more than one character in length. In most cases, use of INPUT #handle, varName works just as well or better for reading devices.

INSTR(a$,b$,n)

INSTR(string1, string2, starting)

Description:

This function returns the position of string2 within string1. If string2 occurs more than once in string 1, then only the position of the leftmost occurance will be returned. If starting is included, then the search for string2 will begin at the position specified by starting.

Usage:

```
print instr("hello there", "lo")

produces:    4

print instr("greetings and meetings", "eetin")

produces:    3

print instr("greetings and meetings", "eetin", 5)

produces:    16
```

If string2 is not found in string1, or if string2 is not found after starting, then INSTR( ) will return 0.

```
print instr("hello", "el", 3)

produces:    0

and so does:

print instr("hello", "bye")
```

INT(n)

Description:

This function removes the fractional part of number, leaving only the whole number part behind.

Usage:

```
[retry]
   input "Enter an integer number>"; i
   if i<>int(i) then bell: print i; " isn't an integer! Re-enter.": goto [retry]
```

KILL s$

```
KILL "filename.ext"
```

Description:

This command deletes the file specified by filename.ext.   The filename can include a complete path specification.

LEFT$(s$, n)

LEFT$(string, number)

Description:

This function returns from string the specified number of characters starting from the left.   So if   string is "hello there", and number is 5, then "hello" would be the result.

Usage:

```
[retry]
  input "Please enter a sentence>"; sentence$
  if sentence$ = "" then [retry]
  for i = 1 to len(sentence$)
    print left$(sentence$, i)
  next i
```

Produces:

```
  Please enter a sentence>That's all folks!
  T
  Th
  Tha
  That
  That'
  That's
  That's_
  That's a
  That's al
  That's all
  That's all_
  That's all f
  That's all fo
  That's all fol
  That's all folk
  That's all folks
  That's all folks!
```

Note: If number is zero or less, then "" (an empty string) will be returned.   If the number is greater than or equal to the number of characters in string, then string will be returned.

See also MID$( ) and RIGHT$( )

LEN(s$)

LEN(string)

Description:

This function returns the length in characters of string, which can be any valid string expression.

Usage:

```
prompt "What is your name?"; yourName$
print "Your name is "; len(yourName$); " letters long"
```

LET var = expr

Description:

LET is an optional prefix for any BASIC assignment expression.   Most do leave the word out of their programs, but some prefer to use it.

Usage:

Either is acceptable:

```
   let name$ = "John"
or
   name$ = "John"
```

Or yet again:

```
   let c = sqr(a^2 + b^2)
or
   c = sqr(a^2 + b^2)
```

LINE INPUT

See INPUT

LOADBMP

LOADBMP "name", "filename.bmp"

Description:

This command loads standard Windows *.BMP bitmap file into Liberty BASIC (see also SAVEBMP). The "name" is a string you would choose to describe the bitmap you're loading, and the "filename.bmp" is the actual name of the bitmap file.   Once loaded, the bitmap can then be displayed in a graphics window type using the drawbmp command (see help file GUI Programming/Window Types/View Graphics Window Commands).

Usage:

See the sample program ttt.bas for an example using LOADBMP.

LOF(#h)

LOF(#handle)

Description:

Returns the # of bytes contained in the file referenced by #handle.

Usage:

```
open "\autoexec.bat" for input as #1
qtyBytes = lof(#1)
for x = 1 to qtyBytes
    print input$(#1, 1) ;
next x
close #1
end
```

LOG(n)

Description:

This function returns the natural log of n.

Usage:

```
print log( 7 )                produces:   1.9459101
```

LOWER$(s$)

Description:

This function returns a copy of the contents of a$, but with all letters converted to lowercase.

Usage:

```
print lower$( "The Taj Mahal" )
```

Produces:

```
the taj mahal
```

LPRINT

LPRINT expr

Description:

This statement is used to send data to the default printer (as determined by the Windows Print Manager).   A series of expressions can follow LPRINT (there does not   need to be any expression at all), each seperated by a semicolon.   Each expression is sent in sequence.   When you are finished sending data to the printer, you should commit the print job by using the DUMP statement.   Liberty BASIC will eventually send your print job, but DUMP forces the job to finish.

Usage:

```
  lprint "hello world"          'This prints hello world
  dump

  lprint "hello ";                 'This also prints hello world
  lprint "world"
  dump

  age = 23
  lprint "Ed is "; age; " years old"      'This prints Ed is 23 years old
  dump
```

Note: see also PRINT, DUMP

MENU

MENU #handle, title,   text, branchLabel,   text, branchLabel, | , . . .

Description:

Adds a pull down menu to the window at #handle. Title specifies the title of the menu, as seen on the menu bar of the window, and each text, branchLabel pair after the title adds a menu item to the menu, and tells Liberty BASIC where to branch to when the menu item is chosen. The | character can optionally be placed between menu items, to cause a seperating line to be added between the items with the menu is pulled down.

As an example, if you wanted to have a graphics window opened, and then be able to pull down some menus to control color and geometric objects, our opening code might look like this:

```
  menu #geo, &Colors, &Red, [setRed], &Green, [setGreen], &Blue, [setBlue]
  menu #geo, &Shapes, &Rectangle, [asRect], &Triangle, [asCircle], &Line, [asLine]
  open "Geometric wite-board" for graphics_nsb as #geo
  input r$   ' stop and wait for a menu item to be chosen
```

Notice that the MENU lines must go before the OPEN statement, and must use the same handle as the window that it will be associated with (#geo in this case). This is the same with the BUTTON statement (see BUTTON). See that execution must be stopped at an input statement for the menu choice to be acted upon. This is also the same as with BUTTON.

Also notice that the & character placed in the title and text items for the menu determines the accelerator placement for each menu. Try experimenting.

MID$( )

MID$(string, index, number)

Description:

Permits the extraction of a sequence of characters from string starting at index. If number is not specified, then all the characters from index to the end of the string are returned. If number is specified, then only as many characters as number specifies will be returned, starting from index.

Usage:

```
print mid$("greeting Earth creature", 10, 5)
```

Produces:

```
Earth
```

And:

```
string = "The quick brown fox jumped over the lazy dog"
for i = 1 to len(string$) step 5
  print mid$(string$, i, 5)
next i
```

Produces:

```
The_q
uick_
brown
_fox_
jumpe␡
d_ove
r_the
_lazy
_dog
```

Note:

See also LEFT$( ) and RIGHT$( )

MKDIR( )

Description:

The MKDIR( ) function attempts to create the directory specified.   If the directory creation is successful the returned value will be 0.   If the directory creation was unsuccessful, a value indicating a DOS error will be returned.

Usage:

```
    'create a subdirectory named temp in the current directory
    result = mkdir( "temp")
    if result <> 0 then notice "Temporary directory not created!"
```

Note: See also RMDIR( )

NAME a$ AS b$

NAME stringExpr1 AS stringExpr2

Description:

This command renames the file specified in the string expression stringExpr1 to stringExpr2. StringExpr1 can represent any valid filename that is not a read-only file, and stringExpr2 can be any valid filename as long as it doesn't specify a file that already exists.

Usage:

```
  'rename the old file as a backup
  name rootFileName$ + ".fre" as rootFileName$ + ".bak"
  'open a new file and write data
  open rootFileName$ + ".fre" for output as #disk
```

NOMAINWIN

Description:

This command instructs Liberty BASIC not to open a main window for the program that includes this statement. Some simple programs which do not use seperate windows for graphics, spreadsheet, or text may use only the main window. Other programs may not need the main window to do their thing, and so simply including NOMAINWIN somewhere in your program source will prevent the window from opening.

If NOMAINWIN is used, when all other windows owned by that program are closed, then the program terminates execution automatically.

It is often better to place a NOMAINWIN statement in your program after it is completed and debugged, so that you can easily terminate an errant program just by closing its main window.

For examples of the usage of NOMAINWIN, examine the included Liberty BASIC programs (buttons1.bas for example).

NOTICE

Description:

This command pops up a dialog box which displays "string expression" and which has a button OK which the user presses after the message is read. Pressing Enter also closes the dialog box.

Two forms are allowed. If "string expression" has no Cr character (ASCII 13), then the title of the dialog box will be 'Notice' and "string expression" will be the message displayed inside the dialog box. If "string expression" does have a Cr character, then the part of "string expression" before Cr will be used as the title for the dialog box, and the part of "string expression" after Cr will be displayed as the message inside.

Usage:

```
notice "Super Stats is Copyright 1992, Mathware"
```

Or:

```
notice "Fatal Error!" + chr$(13) + "The entry buffer is full!"
```

OPEN

OPEN string FOR purpose AS #handle {LEN = #}

Description:

This statement has many functions.   It can be used to open disk files, DLLs, or to open windows of several kinds.

Using OPEN with Disk files:

A typical OPEN used in disk I/O looks like this:

```
  OPEN "\autoexec.bat" for input as #read
```

This example illustrates how we would open the autoexec.bat file for reading.   As you can see, string in this case is "\autoexec.bat", purpose is input, and #handle is read.

string	- this must be a valid pathname. If the file does not exist, it will be created.
purpose	- must be input, output, or random
#handle	- use a unique descriptive word, but must start with a #. This special handle is used to identify the open file in later program statements
LEN = #	- this is an optional addition for use only when opening a random access file. The # determines how many characters long each record in the file is. If this is not specified, the default length is 128 characters. See FIELD, GET, and PUT.

Using OPEN with windows:

A typical OPEN used in windows looks like this:

```
  OPEN "Customer Statistics Chart" for graphics as #csc
```

This example illustrates how we would open a window for graphics.   Once the window is open, there are a wide range of commands that can be given to it.   As you can see, string in this case is "Customer Statistics Chart", which is used as the title of the window, purpose is graphics (open a window for graphics), and the #handle is #csc (derived from Customer Statistics Chart), which will be used as an identifier when sending commands to the window.

string	- can be any valid BASIC string. used to label the window
purpose	- there are a several possibilities here: graphics, spreadsheet, text, window, or dialog (any of these can end in _fs, _nsbars or other suffixes)
#handle	- as above, must be a unique, descriptive word starting with #

Using OPEN with DLLs:

To call functions that reside in a DLL, you must open the DLL first. Additionally, if you want to make a Windows API function call, you must first open the DLL it is contained in. See CALLDLL.

Note: Any opened file or window must be closed before program execution is finished. See CLOSE

Platform$

Description:

This variable holds the string "Windows".   When programming with Liberty BASIC for OS/2, the same variable holds "OS/2".

This is useful so that you can take advantage of whatever differences there are between the two platforms and between the versions of Liberty BASIC.

Note: see also Version$

PLAYWAVE

PLAYWAVE "filename" [, mode]

Description:

Plays a *.wav sound file as specified in filename. If mode is specified, it must be one of the modes specifed below:

sync (or synch)	- wait for the wave file to finish playing (the default)
async (or asynch)	- don't wait for the wave file to finish playing
loop	- play the wave file over and over (cancel with: playwave "")

PRINT

PRINT #handle, pression ; expression(s) ;

Description:

This statement is used to send data to the main window, to a disk file, or to other windows. A series of expressions can follow PRINT (there does not need to be any expression at all), each seperated by a semicolon. Each expression is displayed in sequence. If the data is being sent to a disk file, or to a window, then #handle must be present.

PRINTing to a the main window:

When the expressions are displayed, then the cursor (that blinking vertical bar |) will move down to the next line, and the next time information is sent to the window, it will be placed on the next line down. If you do not want the cursor to move immediately to the next line, then add an additional semicolor to the end of the list of expressions. This prevents the cursor from being moved down a line when the expressions are displayed. The next time data is displayed, it will be added onto the end of the line of data displayed previously.

```
Usage:                         Produces:

   print "hello world"              hello world

   print "hello ";                  hello world
   print "world"

   age = 23
   print "Ed is "; age; " years old"    Ed is 23 years old
```

When sending to a disk file and in regard to the use of the semicolon at the end of the expression list, the rules are similar (only you don't see it happen on the screen). When printing to a window, the expressions sent are usually commands to the window (or requests for information from the window). For more information, see help file GUI Programming.

PROMPT

PROMPT string; responseVar

Description:

The PROMPT statement opens a dialog box, displays string, and waits for the user to type a response and press 'Return' (or press the OK or Cancel button).   The entered information is placed in responseVar.   If Cancel is pressed, then a string of zero length is returned.   If responseVar is set to some string value before PROMPT is executed, then that value will become the 'default' or suggested response.   This means that when the dialog is opened, the contents of responseVar will already be entered as a response for the user, who then has the option to either type over that 'default' response, or to press 'Return' and accept it.

Usage:

```
  .
  .
  response$ = "C:"
  prompt "Search on which Drive? A:, B:, or C:"; response$
[testResponse]
  if response$ = "" then [cancelSearch]
  if len(response$) = 2 and instr("A:B:C:", response$) > 0 then [search]
  prompt "Unacceptable response.   Please try again. A:, B:, or C:"; again$
  goto [testResponse]

[search]
  print "Starting search . . . "
  .
  .
```

PUT #h, n

PUT #handle, n

Description:

PUT is used after a random access file is opened to place a record of information (see FIELD) into the file #handle at record n.   For example:

```
  open "custdata.001" for random as #cust len = 70     ' open a random access file
  field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

  ' enter data into customer variables
  input name$
  .
  .
  ' put the data into record 1
  put #cust, 1

  close #cust
  end
```

Note: See also GET, FIELD

RADIOBUTTON

RADIOBUTTON #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height

Description:

Adds a radiobutton control to the window referenced by #handle.   Radiobuttons have two states, set and reset.   They are useful for getting input of on/off   type information.

All radiobuttons on a given window are linked together, so that if you set one by clicking on it, all the others will be reset.

Here is a description of the parameters of the RADIOBUTTON statement:

"label"	- This contains the visible text of the radiobutton
[set]	- This is the branch label to goto when the user sets the radiobutton by clicking on it.
[reset]	- This is the branch label to goto when the user resets the radiobutton by clicking on it. (this doesn't actually do anything because radiobuttons can't be reset by clicking on them).
xOrigin	- This is the x position of the radiobutton relative to the upper left corner of the window it belongs to.
yOrigin	- This is the y position of the radiobutton relative to the upper left corner of the window it belongs to.
width	- This is the width of the radiobutton control
height	- This is the height of the radiobutton control

Usage:

See the included program radiobtn.bas for an example of how to use radiobuttons.

Note: see also CHECKBOX

REDIM

Description:

Redimensions an already dimensioned array and clears all elements to zero (or to an empty string in the case of string arrays). This can be very useful for writing applications that have data sets of unknown size. If you dimension arrays that are extra large to make sure you can hold data, but then only have a small set of data, then all the space you reserved is wasted. This hurts performance.

Usage:

```
dim cust$(10)   'dimension the array
.
.
.
'now we know there are 510 customers on file
redim cust$(510)
'now read in the customer records
```

REM

REM comment

Description:

The REM statement is used to place comments inside of code to clearly explain the purpose of each section of code. This is useful to both the programmer who writes the code or to anyone who might later need to modify the program. Use REM statements liberally. There is a shorthand way of using REM, which is to use the ' (apostrophe) character in place of the word REM. This is cleaner to look at, but use whichever you prefer. Unlike other BASIC statements, with REM you cannot add another statement after it on the same line using a colon (:) to seperate the statements. The rest of the line becomes part of the REM statement.

Usage:

```
  rem   let's pretend that this is a comment for the next line
  print "The mean average is "; meanAverage
```

Or:

```
  ' let's pretend that this is a comment for the next line
  print "The strength of the quake was "; magnitude
```

This doesn't work:

```
  rem   thank the user : print "Thank you for using Super Stats!"
```

(even the print statement becomes part of the REM statement)

Note:

When using ' instead of REM at the end of a line, the statement seperator :
is not required to seperate the statement on that line from its comment.

For example:

```
  print "Total dollar value: "; dollarValue : rem   print the dollar value
```

Can also be stated:

```
  print "Total dollar value: "; dollarValue   ' print the dollar value
```

Notice that the : is not required in the second form.

RETURN

RETURN

See GOSUB

RIGHT$(s$, n)

RIGHT$(string, number)

Description:

Returns a sequence of characters from the right hand side of string using number to determine how many characters to return.   If   number is 0, then "" (an empty string) is returned.   If number is greater than or equal to the number of characters in string, then string will itself be returned.

Usage:

```
print right$("I'm right handed", 12)
```

Produces:

```
right handed
```

And:

```
print right$("hello world", 50)
```

Produces:

```
hello world
```

Note: See also LEFT$( ) and MID$( )

RND(n)

RND(number)

Description:

This function returns a pseudo random number between 0 and 1.   This can be useful in writing games and some simulations.   The particular formula used in this release might more accurately be called an arbitrary number generator (instead of random number generator), since if a distribution curve of the output of this function were plotted, the results would be quite uneven.   Nevertheless, this function should prove more than adequate (especially for game play).

In MBASIC it makes a difference what the value of parameter number is, but in Liberty BASIC, it makes no difference.   The function will always return an arbitrary number between 0 and 1.

Usage:

```
  ' print ten numbers between one and ten
  for a = 1 to 10
      print int(rnd(1)*10) + 1
  next a
```

RUN s$, mode

RUN stringExpr1 [, mode]

Description:

This command runs external programs.   StringExpr1 should represent the full path and filename of a Windows or DOS executable program, a Liberty BASIC *.TKN file, or a *.BAT file.   This is not a SHELL command, so you must provide the name of a program or batch file, not a DOS command (like DIR, for example).   Execution of an external program does not cause the calling Liberty BASIC program to cease executing.

Here are two examples:

```
    RUN "QBASIC.EXE"      ' run Microsoft's QBASIC

    RUN "WINFILE.EXE", SHOWMAXIMIZED   ' run the File Manager maximized
```

Notice in the second example we can include an additional parameter.   This is because we are running a Windows program.   Here is a list of the valid parameters we can include when running Windows programs:

```
      HIDE
      SHOWNORMAL   (this is the default)
      SHOWMINIMIZED
      SHOWMAXIMIZED
      SHOWNOACTIVE
      SHOW
      MINIMIZE
      SHOWMINNOACTIVE
      SHOWNA
      RESTORE
```

BMPSAVE

BMPSAVE "name", "filename.bmp"

Description:

This command saves a named Liberty BASIC bitmap to a standard Windows *.BMP bitmap file (see also LOADBMP). The "name" is a string naming the bitmap you are saving, and the "filename.bmp" is the actual name of the bitmap file.

SCAN

Description:

The SCAN statement causes Liberty BASIC to stop what it is doing for a moment and process Windows keyboard and mouse messages.   This is useful for any kind of routine that needs to run continuously but which still needs to process button clicks and other actions.   In this way, SCAN can be used as an INPUT statement that doesn't stop and wait.

Example:

```
    'scan example - digital clock

    nomainwin

    WindowWidth = 120
    WindowHeight = 95
    statictext #clock.time, "xx:xx:xx", 15, 10, 90, 20
    button #clock.12hour, "12 Hour", [twelveHour], UL, 15, 40, 40, 20
    button #clock.24hour, "24 Hour", [twentyfourHour], UL, 60, 40, 40, 20
    open "Clock" for window_nf as #clock
    print #clock, "trapclose [quit]"
    print #clock.time, "!font courier_new 8 15"
    print #clock.12hour, "!font ariel 5 11"
    print #clock.24hour, "!font ariel 5 11"

    goto [twelveHour]

[timeLoop]

    if time$ <> time$() then
        time$ = time$()
        gosub [formatTime]
        print #clock.time, formattedTime$
    end if

    scan      'check for user input

    goto [timeLoop]

[formatTime]

    hours = val(left$(time$, 2))

    if twelveHourFormat = 1 then
        if hours > 12 then
            hours = hours - 12
            suffix$ = " PM"
        else
            if hours = 0 then hours = 12
            suffix$ = " AM"
        end if
    else
        suffix$ = ""
    end if

    formattedTime$ = prefix$+right$("0"+str$(hours), 2)+mid$(time$, 3)+suffix$

    return


[twelveHour]   'set up twelve-hour mode

    twelveHourFormat = 1
    time$ = ""
    prefix$ = ""
    goto [timeLoop]

[twentyfourHour]   'set up twentyfour-hour mode

    twelveHourFormat = 0
    time$ = ""
    prefix$ = " "
    goto [timeLoop]

[quit]   'exit our clock

    close #clock
    end
```

SIN(n)

Description:

This function return the sine of n.

Usage:

```
.
.
for t = 1 to 45
   print "The sine of "; t; " is "; sin(t)
next t
.
.
```

Note: See also COS( ) and TAN( )

SORT

SORT arrayName(, start, end, [column]

Description:

This command sorts both doubleand single dimensioned arrays. Arrays can be sorted in part or in whole, and with double dimensioned arrays, the specific column to sort by can be declared. When this option is used, all the rows are sorted against each other according to the items in the specified column.

Usage:

Here is the syntax for the sort command:

```
    sort arrayName(, i, j, [,n]
```

This sorts the array named arrayName(starting with element i, and ending with element j. If it is a double dimensioned array then the column parameter tells which nth element to use as a sort key. Each WHOLE row moves with its corresponding key as it moves during the sort. So let's say you have a double dimensioned array holding sales rep activity:

```
    repActivity(x, y)
```

So you're holding data, one record per x position, and your record keys are in y. So for example:

```
    repActivity(1,1) = "Tom Maloney" : repActivity(1,2) = "01-09-93"
    repActivity(2,1) = "Mary Burns" : repActivity(2,2) = "01-10-93"
     .
     .
     .
    repActivity(100,1) = "Ed Dole" : repActivity(100,2) = "01-08-93"
```

So you want to sort the whole 100 items by the date field. This is how the command would look:

```
    sort repActivity(, 1, 100, 2
```

If you wanted to sort by name instead, then change the 2 to a 1, like this:

```
    sort repActivity(, 1, 100, 1
```

STR$(n)

Description:

This function returns a string expressing the result of numericExpression. In MBASIC, this function would always return a string representation of the expression and it would add a space in front of that string. For example in MBASIC:

```
   print len(str$(3.14))
```

Would produce the number 5 (a space followed by 3.14 for a total of 5 characters).

Liberty BASIC leaves it to you to decide whether you want that space or not. If you don't want it, then you need not do anything at all, and if you do want it, then this expression will produce the same result under Liberty BASIC:

```
   print len(" " + str$(3.14))
```

Usage:

```
   .
   .
[kids]
   ' use str$( ) to validate entry
   input "How many children do you have?"; qtyKids
   qtyKids$ = str$(qtyKids)
   ' if the entry contains a decimal point, then the response is no good
   if instr(qtyKids$, ".") > 0 then print "Bad response. Reenter." : goto [kids]
```

TAN(n)

Description:

This function returns the tangent of n.

Usage:

```
.
.
for t = 1 to 45
   print "The tangent of "; t; " is "; tan(t)
next t
.
.
```

Note: See also SIN( ) and COS( )

TIME$( )

Description:

This function returns a string representing the current time of the system clock in 24 hour format.   This function replaces the time$ variable used in MBASIC.   See also DATE$( ).

Usage:

```
' display the opening screen
print "Main selection screen             Time now: "; time$( )
print
print "1. Add new record"
print "2. Modify existing record"
print "3. Delete record"
```

TRACE n

TRACE number

Description:

This statement sets the trace level for its application program.   This is only effective if the program is run using the Debug menu selection (instead of RUN).   If Run is used, then any TRACE statements are ignored.

There are three trace levels: 0, 1, and 2.   Here are the effects of these levels:

0 = single step mode or STEP
1 = animated trace or WALK
2 = full speed no trace or RUN

When any Liberty BASIC program first starts under Debug mode, the trace level is always initially 0. You can then click on any of the three buttons (STEP, WALK, RUN) to determine what mode to continue in.   When a TRACE statement is encountered, the trace level is set accordingly, but you can recover from this new trace level by clicking again on the desired button.

If you are having trouble debugging code at a certain spot, then you can add a TRACE statement (usually level 0) just before that location, run in Debug mode and then click on RUN.   When the TRACE statement is reached, then the debugger will kick in at that point.

Usage:

```
'Here is the trouble spot
trace 0  ' kick down to single step mode
for index = 1 to int(100*sin(index))
  print #graph, "go "; index ; " "; int(100*cos(index))
next index
```

TRIM$(s$)

TRIM$(stringExpression)

Description:

This function removes any spaces from the start and end of the string in stringExpression.   This can be useful for cleaning up data entry among other things.

Usage:

```
  sentence$ = "  Greetings  "
  print len(trim$(sentence$))
```

Produces: 9

UPPER$(s$)

Description:

This function returns a copy of the contents of a$, but with all letters converted to uppercase.

Usage:

```
print upper$( "The Taj Mahal" )
```

Produces:

```
THE TAJ MAHAL
```

USING( )

USING(templateString, numericExpression)

Description:

This function formats numericExpression as a string using templateString. The rules for the format are like those in Microsoft BASIC's PRINT USING statement, but since using( ) is a function, it can be used as part of a larger BASIC expression instead of being useful only for output directly.

Unlike PRINT USING, the using( ) function does not round numbers.

Usage:

```
' print a column of ten justified numbers
for a = 1 to 10
    print using("####.##",  rnd(1)*1000)
next a
```

VAL(s$)

VAL(stringExpression)

Description:

This function returns a numeric value for stringExpression is stringExpression represents a valid numeric value or if it starts out as one.   If not, then zero is returned.   This function lets your program take string input from the user and carefully analyze it before turning it into a numeric value if and when appropriate.

Usage:

```
print 2 * val("3.14")          Produces:        6.28

print val("hello")             Produces:        0

print val("3 blind mice")      Produces:        3
```

Version$

Version$

Description:

This variable holds the version of Liberty BASIC, in this case "1.2".

This is useful so that you can take advantage of whatever differences there are between the different versions of Liberty BASIC.

Note: see also Platform$

WHILE...WEND

WHILE expression
  {some code}
WEND

Description:

These two statements comprise the start and end of a control loop. Between the WHILE and WEND statements place code (optionally) that will be executed repeatedly while expression evaluates to true. The code between any WHILE statement and its associated WEND statement will not execute even once if the WHILE expression initially evaluates to false. Once execution reaches the WEND statement, for as long as the WHILE expression evaluates to true, then execution will jump back to the WHILE statement. Expression can be a boolean, numeric, or string expression.

Usage:

```
  ' loop until midnight (go read a good book)
  while time$ <> "00:00:00"
      ' some action performing code might be placed here
  wend
```

Or:

```
  ' loop until a valid response is solicited
  while val(age$) = 0
     input "How old are you?"; age$
     if val(age$) = 0 then print "Invalid response.  Try again."
  wend
```

Note: DO NOT break out of a WHILE...WEND loop using GOTO. If you do, then your Liberty BASIC program may behave unpredictably.

You may GOSUB out of a WHILE...WEND loop, because when your subroutine finishes, it will RETURN to the loop. The following example is an illustration of what you SHOULD NOT DO.

```
  while count < 10
    input "Enter a name (or a blank line to quit) ?"; n$
    if n$ = "" then [exitLoop]
    list$(count) = n$
    count = count + 1
  wend
[exitLoop]
```

Instead, you might do this:

```
  while count < 10
    input "Enter a name (or a blank line to quit) ?"; n$
    if n$ = "" then
        count = 10
      else
        list$(count) = n$
        count = count + 1
      end if
```

wend

WORD$(s$, n)

WORD$(stringExpression, n)

Description:

This function returns the nth word in stringExpression.   The leading and trailing spaces are stripped from stringExpression and then it is broken down into 'words' at the remaining spaces inside.   If n is less than 1 or greater than the number of words in stringExpression, then "" is returned.

Usage:

```
  print word$("The quick brown fox jumped over the lazy dog", 5)
```

Produces:

```
  jumped
```

And:

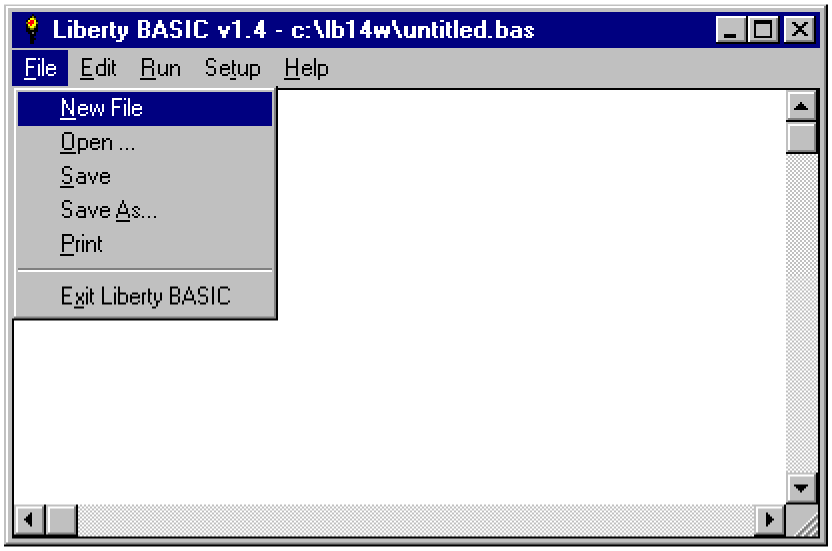
```
  ' display each word of sentence$ on its own line
  sentence$ = "and many miles to go before I sleep."
  token$ = "?"
  while token$ <> ""
      index = index + 1
      tokens$ = word$(sentence$, index)
      print token$
  wend
```

Produces:

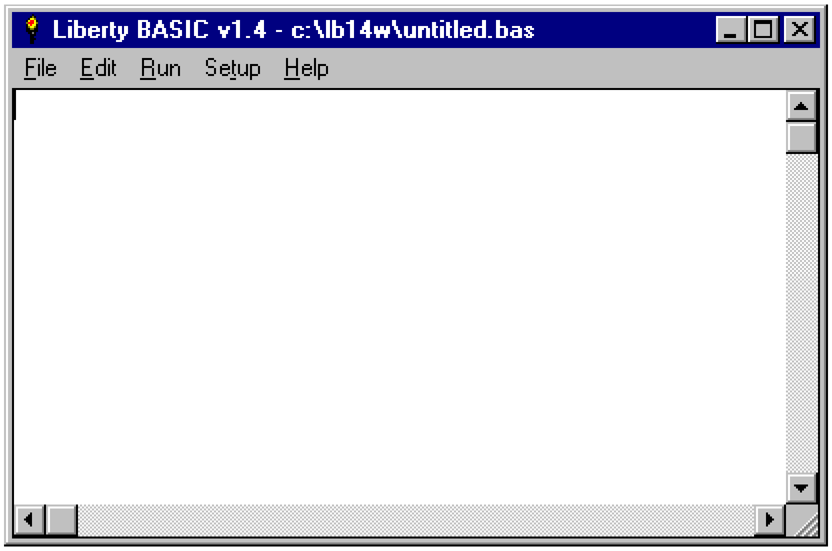
```
  and
  many
  miles
  to
  go
  before
  I
  sleep.
```

Writing your own Programs

Let's write a short BASIC program.   Pull down the File menu and select New File.



If you are ask if you want to save changes, click on No.   Now you will see this:

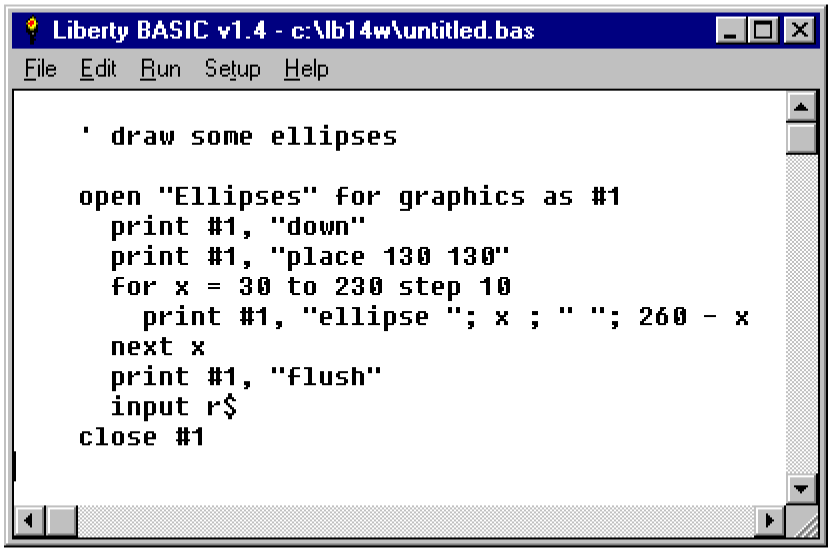


Now let's type a short program into our editor window.   Type carefully (especially notice spaces between double quotes).

```
    ' draw some ellipses

    open "Ellipses" for graphics as #1
      print #1, "down"
      print #1, "place 130 130"
      for x = 30 to 230 step 10
```

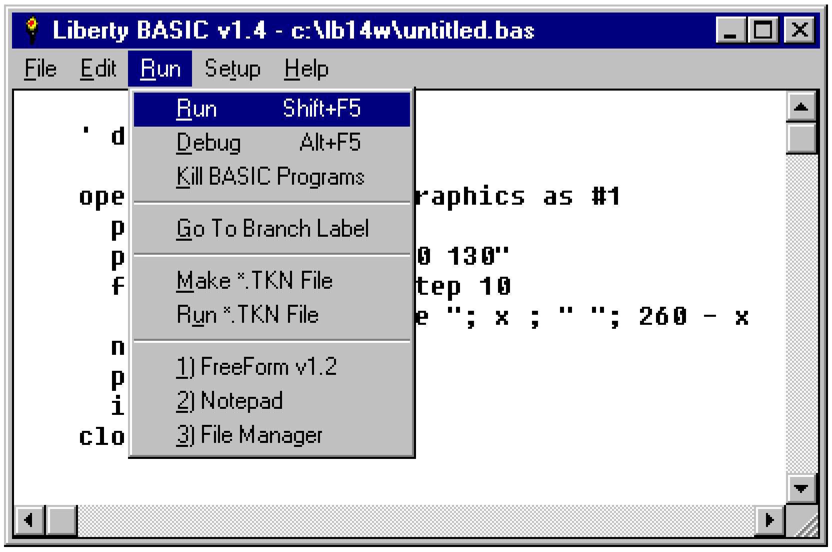
```
        print #1, "ellipse "; x ; " "; 260 - x
      next x
      print #1, "flush"
      input r$
    close #1
```

You should see something like this:

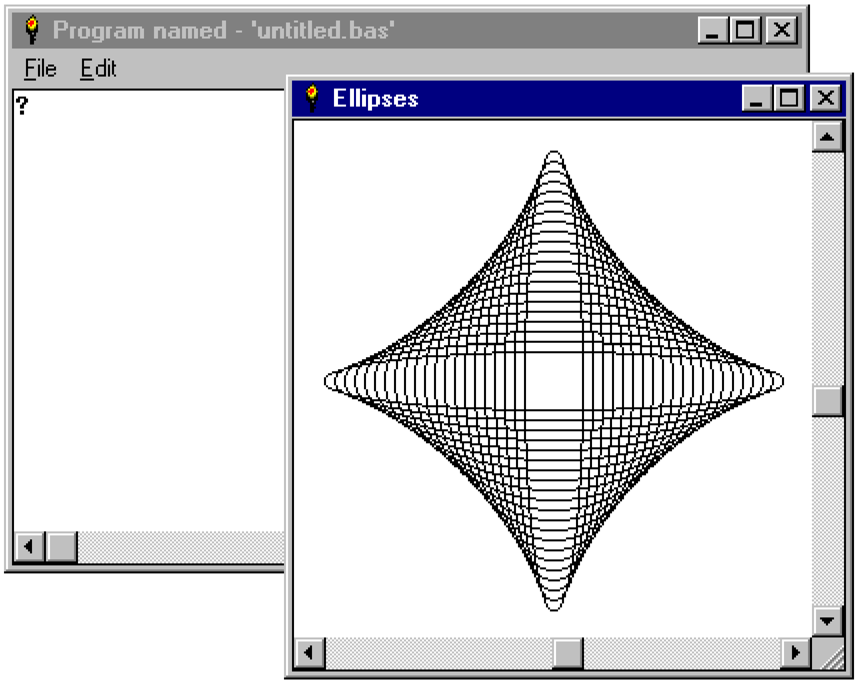


```
    ' draw some ellipses

    open "Ellipses" for graphics as #1
      print #1, "down"
      print #1, "place 130 130"
      for x = 30 to 230 step 10
        print #1, "ellipse "; x ; " "; 260 - x
      next x
      print #1, "flush"
      input r$
    close #1
```

Now pull down the Run menu and select Run, as shown.

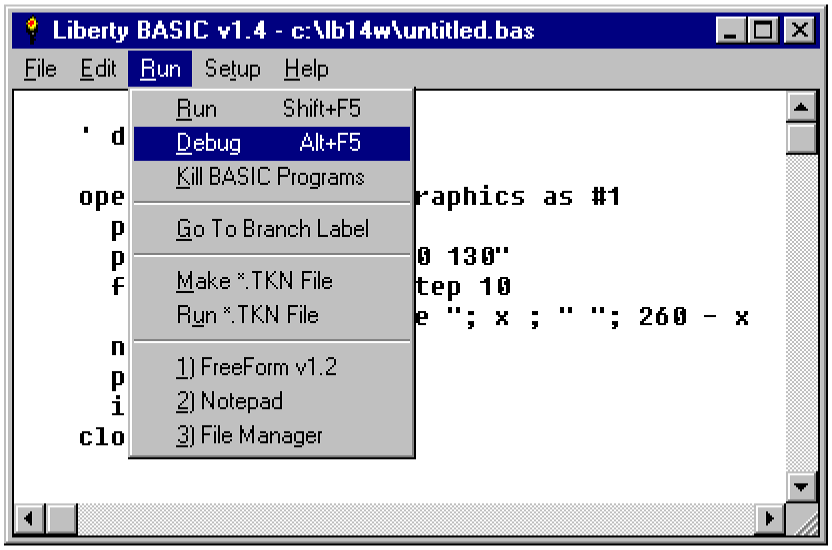


The program will run as seen below.   Now close both of the new windows.


Program named - 'untitled.bas'
File
Edit
?
Ellipses

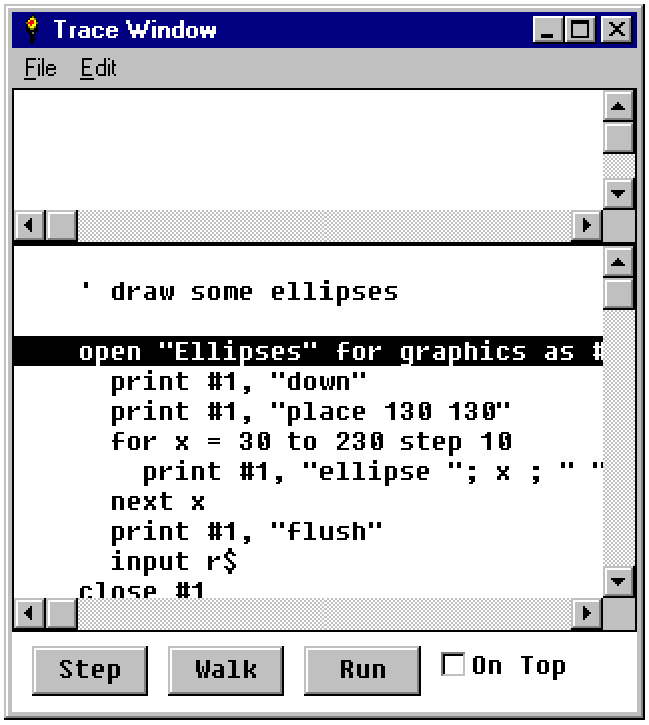
Using the Debugger

Let's take a closer look at how our program works using the debugger (see previous section). Pull down the Run menu and select Debug.



```
    ' d
    ope                              raphics as #1
      p
      p                              0 130"
      f                              tep 10
                                     e "; x ; " "; 260 - x
      n
      p
      i
    clo
```

A Trace Window will appear, and also another window labeled Program named - 'untitled.bas'



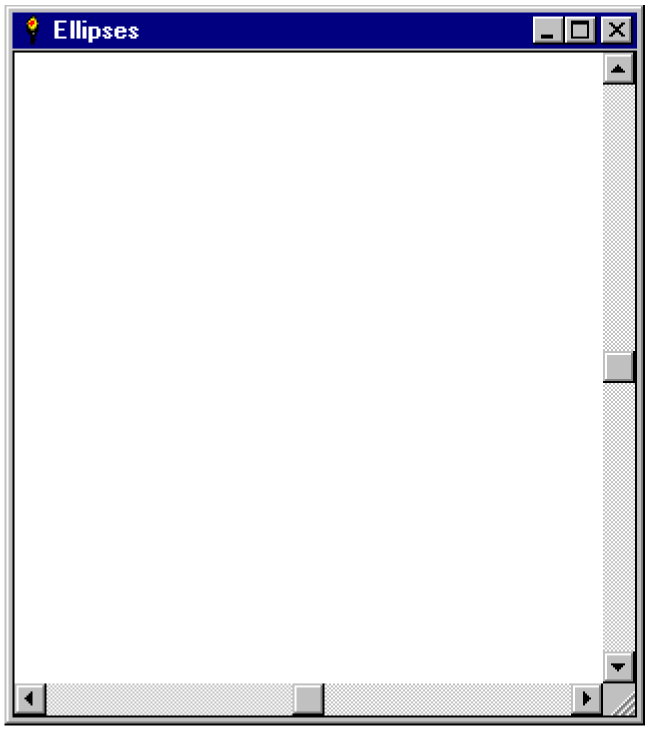
```
  ' draw some ellipses

open "Ellipses" for graphics as #
  print #1, "down"
  print #1, "place 130 130"
  for x = 30 to 230 step 10
    print #1, "ellipse "; x ; " "
  next x
  print #1, "flush"
  input r$
close #1
```

Select the Trace Window to bring it to the foreground and to make it the active window. Notice that it has two panes. The pane on the top shows variables as they change value. The pane on the bottom shows each line of code as it executes.   The three buttons on the bottom of the window let you pick three different modes of execution:

Step	- Step one line at a time through program execution
Walk	- Run the program non stop highlighting each line as it executes
Run	- Run full speed. Do not highlight each line

Execution always begins in Step mode when the Debug option is used.

Now let's click on the Step button once.   Now notice that the Trace Window now highlights the next line, and that a graphics window appears labeled Ellipses.

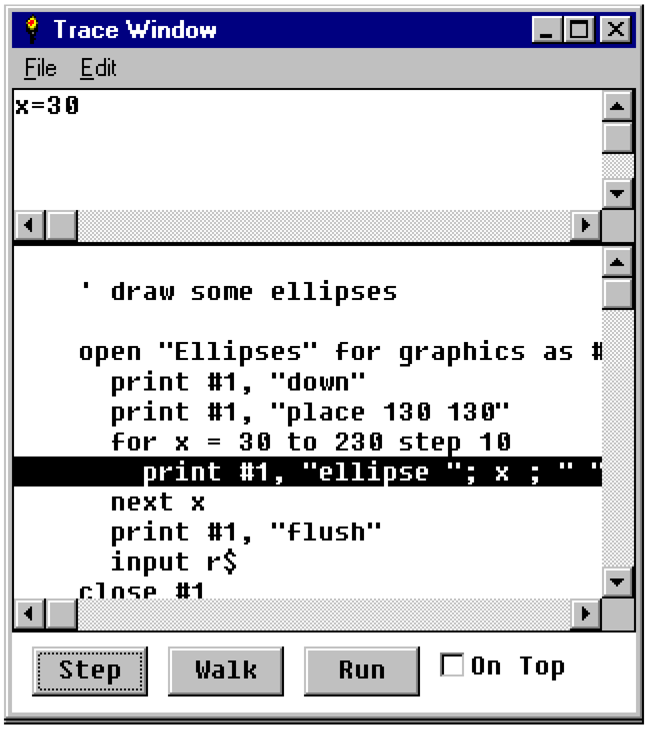


Click on the trace window to bring it to the front, and click on Step twice more.   The two statements below will be executed:

```
print #1, "down"
print #1, "place 130 130"
```

You won't be able to immediately see the effect of thesetwo statements.   The first one tells the window's graphic pen to be 'lowered' to the surface of its 'paper'.   The second statement places the pen at 130 in x and y.

Now click on Step again.   Now look at the variables pane in the Trace Window.



```
x=30
```

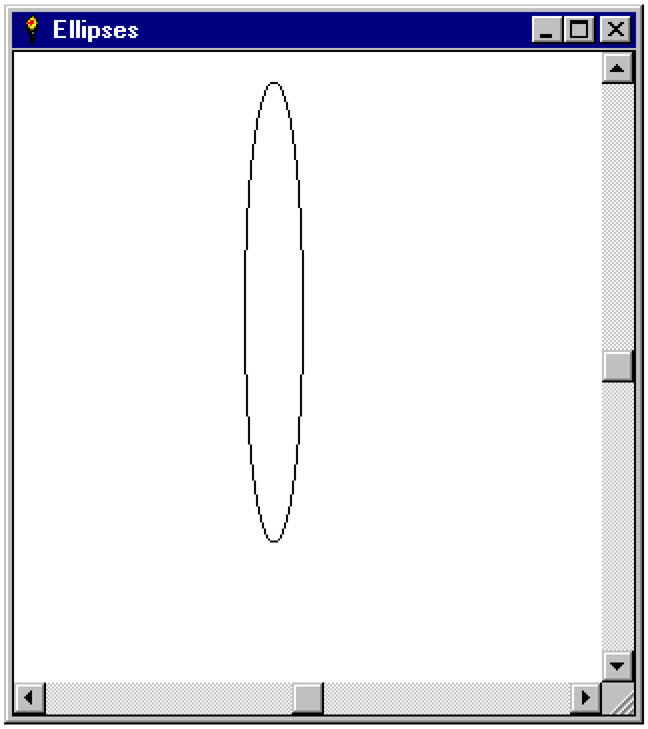
```
  ' draw some ellipses

  open "Ellipses" for graphics as #
    print #1, "down"
    print #1, "place 130 130"
    for x = 30 to 230 step 10
      print #1, "ellipse "; x ; " "
    next x
    print #1, "flush"
    input r$
  close #1
```

This shows that the variable x has been assigned the value 30. Each and every time that x (or any other variable), changes, we will be informed as to just what that change is.

Now click on Step again. The line:

```
    print #1, "ellipse "; x ; " "; 260 - x
```

will be executed, and you will see this:



Now click on Step a dozen or so times, watching the value of x change and seeing several new ellipses drawn. Finally, click on Walk and the program will run non-stop, highlighting each line as it goes, and displaying each new value of x. When this is done, you may close the trace window. Liberty BASIC will ask if you want to terminate ellipses.bas. Respond by pressing Enter or clicking on Yes. Liberty BASIC will close the other two windows automatically (the graphics window with our ellipses, and the window labeled: Program named: 'untitled.bas').

RMDIR( )

Description:

The RMDIR( ) function attempts to remove the directory specified.   If the directory removal is successful the returned value will be 0.   If the directory removal was unsuccessful, a value indicating a DOS error will be returned.

Usage:

```
'remove a subdirectory named "pigseye" in the current root directory
result = rmdir( "\pigseye")
if result <> 0 then notice "Temporary directory not removed!"
```

Note: See also MKDIR( )

CURSOR

Description:

The CURSOR command was added to make it easy to change mouse pointers one of 5 predefined shapes.

NORMAL	= the default pointer
ARROW	= the standard Windows arrow
CROSSHAIR	= a + shaped pointer
HOURGLASS	= the Windows hourglass
TEXT	= the text insertion I-beam

Example:

```
cursor hourglass
for i = 1 to n
    'perform some work
next i
cursor normal
```

Your program's code is responsible to setting the cursor back to the default (normal) when appropriate. If a runtime error halts your program, the cursor will automatically revert to normal.

GUI Programming

Don't forget to check out the GUI Programming section our included course!

A Simple Example

In Liberty BASIC windows are treated like files, and we can refer to anything in this class as a BASIC 'Device'. To open a window we use the OPEN statement, and to close the window we use the CLOSE statement. To control the window we 'print' to it, just as we would print to a file. The commands are sent as strings to the device. As a simple example, here we will open a graphics window, center a pen (like a Logo turtle), and draw a simple spiral. We will then pause by opening a simple dialog. When you confirm the exit, we will close the window:

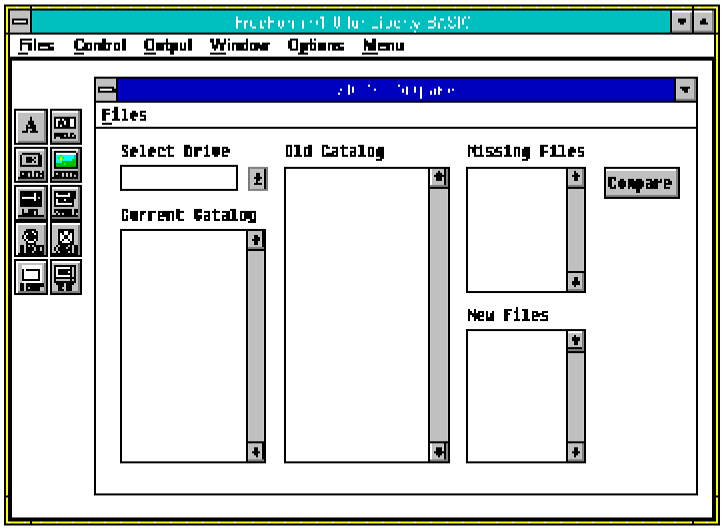
```
    button #graph, Exit, [exit], LR, 35, 20  'window will have a button
    open "Example" for graphics as #graph    'open graphics window
    print #graph, "up"                      'make sure pen is up
    print #graph, "home"                    'center the pen
    print #graph, "down"                    'make sure pen is down
    for index = 1 to 30                     'draw 30 spiral segments
      print #graph, "go "; index            'go foreward 'index' places
      print #graph, "turn 118"              'turn 118 degrees
    next index                              'loop back 30 times
    print #graph, "flush"                   'make the image 'stick'

  [inputLoop]
    input b$ : goto [inputLoop]             'wait for button press

  [exit]
    confirm "Close Window?"; answer$        'dialog to confirm exit
    if answer$ = "no" then [inputLoop]      'if answer$ = "no" loop back
    close #graph

  end
```

Using FreeForm

Using the information in this help file, you could manually create all the code you need to create the windows you need for your programs. This is a lot of work, time, and tedium. FreeForm was created to end this drudgery by letting you visually draw your program's windows, and it writes the Liberty BASIC source code for you.

By selecting the Run menu and picking FreeForm Lite from that menu, FreeForm will be loaded.

Below we see an example run of FreeForm.



The menu items for FreeForm are:

- Files	For loading and saving forms
- Control	For inspecting properties, deleting, and moving controls
- Output	For generating Liberty BASIC code from forms
- Window	For changing the form window handle, title, and type
- Options	For setup and configuration
- Menu	For editing the forms menu bar

The ten buttons on the left side of the FreeForm window represent different kinds of controls that can be added to a form. When a button is clicked on, you are optionally asked for some information (a button label perhaps), and the control is added to the window. Then you can move the control and size it as desired.

Make sure to inspect each control (double-click on a control to inspect it) to set its properties.

When you are happy with the layout of the window, pull down the Output menu and select Produce Code. This will open a window and generate Liberty BASIC code into that windows. You can copy this code into your programs. Selecting the menu item Produce Code + Outline produces even more code for you if desired.

NOTE: There are some features of Liberty BASIC that are not supported in the current release of FreeForm. Most notably, some window types (see the section Window Types in this help file for a complete list of supported types).

Sizing and Placement of Windows

Click here for Sizing and Placement of Dialog Boxes.

The size and placement of any non-dialog window can be easily determined before it is opened in Liberty BASIC. If you do choose not to specify the size and placement of the windows that your programs open, Liberty BASIC will pick default sizes. However, for effect it is often best that you exercise control over this matter.

There are four special variables that you can set to select the size and placement of your windows, whether they be text, graphics, or spreadsheet:

UpperLeftX, UpperLeftY, WindowWidth, and WindowHeight

You can also get the width and the height of the display screen with these variables:

DisplayWidth and DisplayHeight

Set UpperLeftX and UpperLeftY to the number of pixels from the upper-left corner of the screen to position the window. Often determining the distance from the upper-left corner of the screen is not as important as determining the size of the window.

Set WindowWidth and WindowHeight to the number of pixels wide and high that you want the window to be when you open it.

Once you have determined the size and placement of your window, then open it. Here is an example:

```
[openStatus]

    UpperLeftX = 32
    UpperLeftY = 32
    WindowWidth = 190
    WindowHeight = 160

    open "Status Window" for spreadsheet as #stats
```

This will open a window 32 pixels from the corner of the screen, and with a width of 190 pixels, and a height of 160 pixels.

Sizing and Positioning Dialog Boxes

Liberty BASIC uses the WindowWidth and WindowHeight variables to size dialog boxes in the same fashion as with other window types (click here for info).   On the other hand, Liberty BASIC dynamically places dialog boxes when they open so that the upper-left corner of the window is at the current mouse position (unless this would cause some part of the dialog box to disappear off the side of the display).

The answer to this if you want to control the opening position of any dialog box is to use Windows API calls to:

1) Get the current position of the mouse
2) Place the mouse where you want the dialog box to open
3) Open the dialog box
4) Put the mouse back where it was

Here's some example code to do this:

```
'This code snippet shows how to position a dialog
'box when it is opened.  In this case, we will center
'it on the screen.

'define structures
struct winRect, _
    orgX as uShort, _
    orgY as uShort, _
    extentX as uShort, _
    extentY as uShort

struct point, _
    x as short, _
    y as short

'open USER.DLL to make API calls
open "user.dll" for dll as #user

'get the current cursor position
calldll #user, "GetCursorPos", _
    point as struct, _
    result as void
x = point.x.struct
y = point.y.struct

'let's do some math to figure out where our window belongs
WindowWidth = 300
WindowHeight = 200
topLeftX = int((DisplayWidth - WindowWidth) / 2)
topLeftY = int((DisplayHeight - WindowHeight) / 2)

'put the cursor where the origin of the window should be
calldll #user, "SetCursorPos", _
    topLeftX as ushort, _
    topLeftY as ushort, _
    result as void

'open the dialog box
open "My Dialog Box" for dialog as #main

'put the cursor back where it was
calldll #user, "SetCursorPos", _
```

```
    x as ushort, _
    y as ushort, _
    result as void

'close USER.DLL
close #user

'wait here
input r$

close #main

end
```

Week One - An introduction to Liberty BASIC

It may sound incredible, but anyone can program for Microsoft Windows. Creating software for Microsoft Windows was once the domain of the elite C and C++ programmer. But finally, after years of waiting, tools have arrived for the non-guru computer user. This tutorial is written about one of these tools, Liberty BASIC, and about how to make it work for you.

What is Liberty BASIC?

Liberty BASIC is a Windows programming tool that brings BASIC's ease of use to Microsoft Windows.

Liberty BASIC includes:

> A powerful BASIC language for Windows;
> A Visual Development Tool, Freeform;
> An editor for writing BASIC programs;
> An easy to use tracing debugger;
> Easy calling of DLLs and APIs;
> A programmable spreadsheet;
> Color graphics capability

An OS/2 version is also available!!!

Next Section: Overview of this Tutorial

Overview of this Tutorial

This tutorial covers:

Next Section: Programming: What is it?

Programming: What is it?

There's nothing mystical about programming computers. Although the newest software on the market today begins to look like magic, all software is built from the ground up out of combinations of the simplest software parts. Once you learn what these software parts are and how they're used, hard work and imagination can take you almost anywhere.

Programming is (simply put) the laying out of simple steps to solve a problem, and in a way that a computer can understand. This is a little bit like teaching a person. These steps must be arranged in the correct order.

For example:

How to drive a car with automatic transmission:

```
    Get into drivers seat ;
    Fasten safety belt ;
    Insert ignition key and turn it to start engine ;
    Press brake with foot ;
    Move transmission selection to D ;
    Look around to see if you're safe ;
    Remove foot from brake ;
    Press accelerator pedal with foot ;
    Manuever into traffic ;
    Don't crash
```

Obviously if the above steps are scrambled up (and maybe even if they aren't) you're in for a pretty big insurance claim. Not only that, but if the instructions are given to someone who speaks only, say, Chinese, we will have a similarly spectacular crash! In the same way, computers are particular about both the order and content of the instructions we give them.

A program in its simplest form usually contains three kinds of activity:

INPUT	The program asks the user for some kind of information ;
CALCULATION	The program transforms or manipulates the information ;
OUTPUT	The program displays the final result of CALCULATION

It is the programmer's job to determine exactly how to accomplish these steps.

Next Section: An Introduction to BASIC

An Introduction to BASIC

BASIC (Beginners All purpose Symbolic Instruction Code) was created in the 1960's as an easy to learn programming language for computers. Because of BASIC's simple form and because it was an interpreted language and gave the programmer instant feedback, it became the most popular programming language when microcomputers made their debut, and it has a large following even today.

This tutorial introduces the first principles of Liberty BASIC, but doesn't provide a thorough description of all language features. For more on the full language and command set, refer to the documentation included with your copy of Liberty BASIC.

Salestax.bas, a simple BASIC program.

Now let's create a very simple program to introduce you to the simplest of BASIC's features. We want a BASIC program that:

1 - Asks for a dollar and cent amount for goods ;
2 - Calculates a 5% sales tax amount for the dollar and cent figure ;
3 - Displays the tax amount the total amount

INPUT - First we need an instruction for the computer that gets information from the user. In BASIC there are several ways to do this but we will choose the input command for our program. In this case input would be used like so:

```
  input "Type a dollar and cent amount ?"; amount
                                           ^------this is a variable
```

This line of BASIC code will display the words, "Type a dollar and cent amount ?", and the computer will stop and wait for the user to type something in. When the [Enter] key is pressed, the typed information will then be stored in the variable* amount.

*variable - In programming, you must assign each bit of data (or information) a unique name. This combination of a name and its data is called a variable because the data part can vary each time the program is used. When you edit a program, you choose a name for each variable. You pick the name for each variable to best fit the kind of data it represents. BASIC doesn't care what names you assign to your variables, except you can't use the names of BASIC commands or functions (called reserved words). Choose names that make it easy for anyone to understand what the BASIC program code means and does. When running a program, BASIC uses the data part of the variable in its calculations. BASIC uses the name of the variable to fetch its data part or to store new data in that variable. Variable data can change many times during the execution of a BASIC program.

CALCULATION - Now we need to calculate the tax for the data in our amount variable:

```
     let tax = amount * 0.05
```

This line of code creates a new variable called "tax" to hold our computed tax data. The BASIC command, "let", tells BASIC to calculate the arithmetic on the right side of the = and set the data of the variable "tax" equal the results of the equation. The "let" word is optional (and most programmers leave it out) but I use it here as an example of BASIC syntax. It could have been coded as follows:

```
     tax = amount * 0.05
```

Now you may be wondering what is that funny little '*' (called asterisk). Since there are no formal arithmatic symbols on a typewriter keyboard, most programming languages use * to denote multiplication, / for division, and the addition and subtraction symbols get lucky and are + and - (what else?).

OUTPUT - Now that we have calculated our tax amount, we will display it with:

```
print "Tax is: "; tax; ". Total is: "; tax + amount
```

The print command displays the information to the screen. The line of code above shows how print is used to display several items of data, each separated by a ';' (semicolon).

The items are:

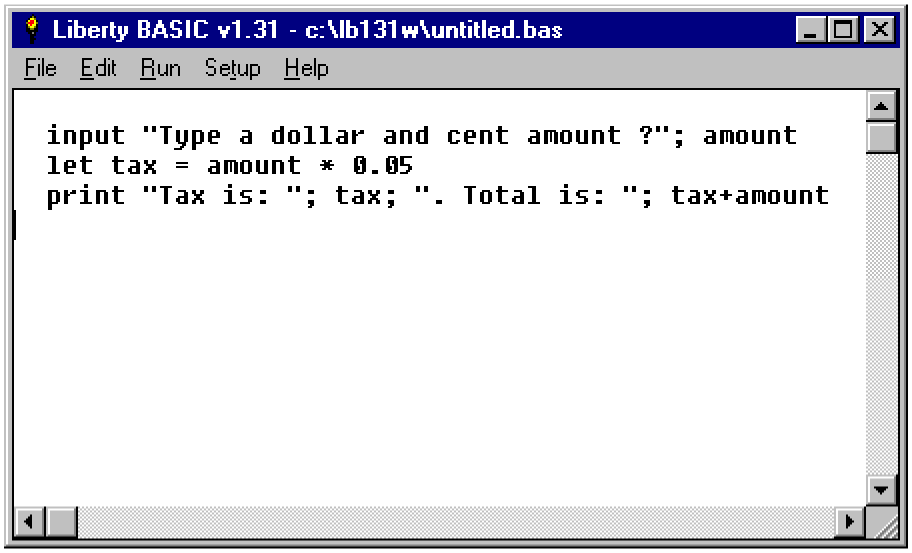
`"Tax is: "`	- This displays on screen as is, but without the quotation marks ;
`tax`	- This displays the value of the variable tax ;
`". Total is: "`	- This also displays on screen as is, but without quotation marks ;
`tax + amount`	- This displays the sum of the two variables, tax and amount

These will all be displayed on the same line. The semicolons are not displayed. Each print command is followed by a carriage return. The result might look like this:

```
Tax is: 0.05. Total is: 1.05
```

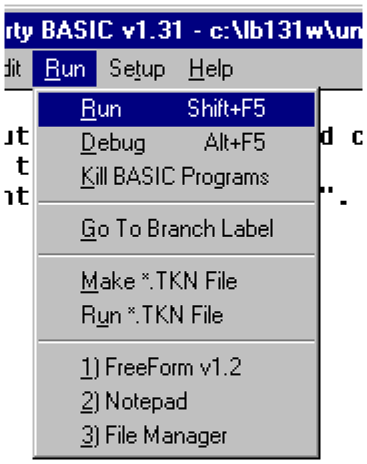
Now let's run the program. Type or cut/paste the following program into the Liberty BASIC editor so that it looks as shown.

```
    input "Type a dollar and cent amount "; amount
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax+amount
```

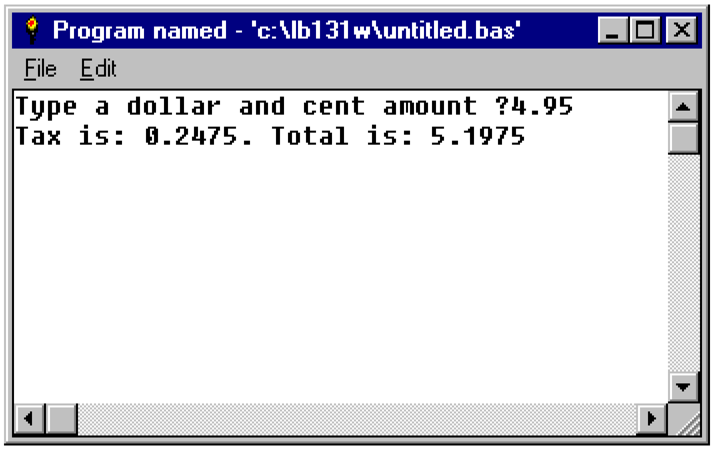


```
input "Type a dollar and cent amount ?"; amount
let tax = amount * 0.05
print "Tax is: "; tax; ". Total is: "; tax+amount
```

Now run the program ...



And here is a sample run:



```
Type a dollar and cent amount ?4.95
Tax is: 0.2475. Total is: 5.1975
```

Now let's save our program. Select the File menu and choose Save As. Now type the name salestax.bas and click on OK.

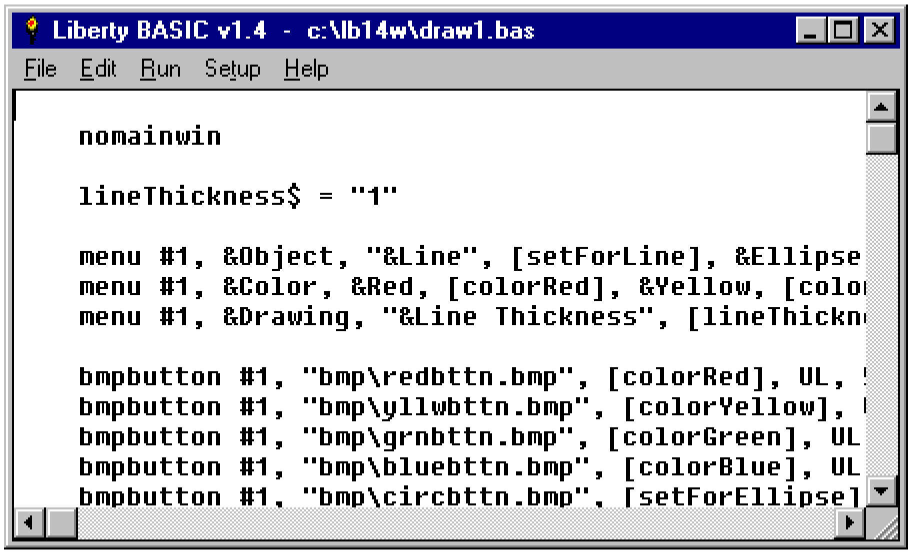
Next Section: Goto - Doing something more than once

Creating a tokenized file

By making a *.tkn file from a *.bas source file, we now have a file that:

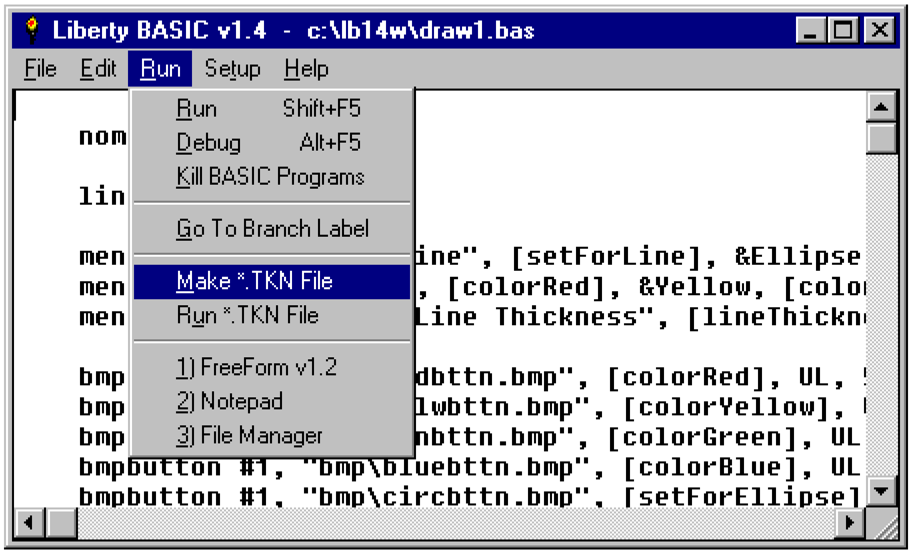
- starts up much faster (very important for large files)
- can be distributed royalty-free using Liberty BASIC's runtime engine (registered version only)
- can be added to the Liberty BASIC's Run menu as an external program (and run instantly by selecting it from that menu).

Let's create a *.tkn file from one of our sample programs. Let's reopen our drawing program. See below.

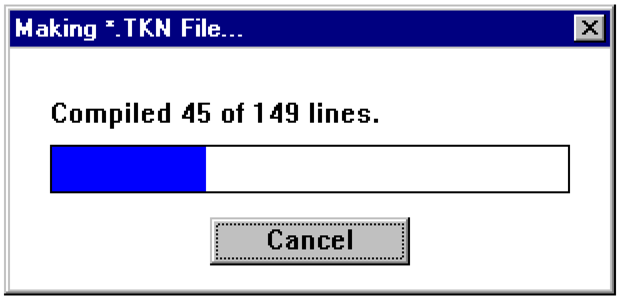


```
    nomainwin

    lineThickness$ = "1"

    menu #1, &Object, "&Line", [setForLine], &Ellipse
    menu #1, &Color, &Red, [colorRed], &Yellow, [colo
    menu #1, &Drawing, "&Line Thickness", [lineThickn

    bmpbutton #1, "bmp\redbttn.bmp", [colorRed], UL,
    bmpbutton #1, "bmp\yllwbttn.bmp", [colorYellow],
    bmpbutton #1, "bmp\grnbttn.bmp", [colorGreen], UL
    bmpbutton #1, "bmp\bluebttn.bmp", [colorBlue], UL
    bmpbutton #1, "bmp\circbttn.bmp", [setForEllipse]
```

Now pull down the Run menu and select Make *.TKN File, like so:

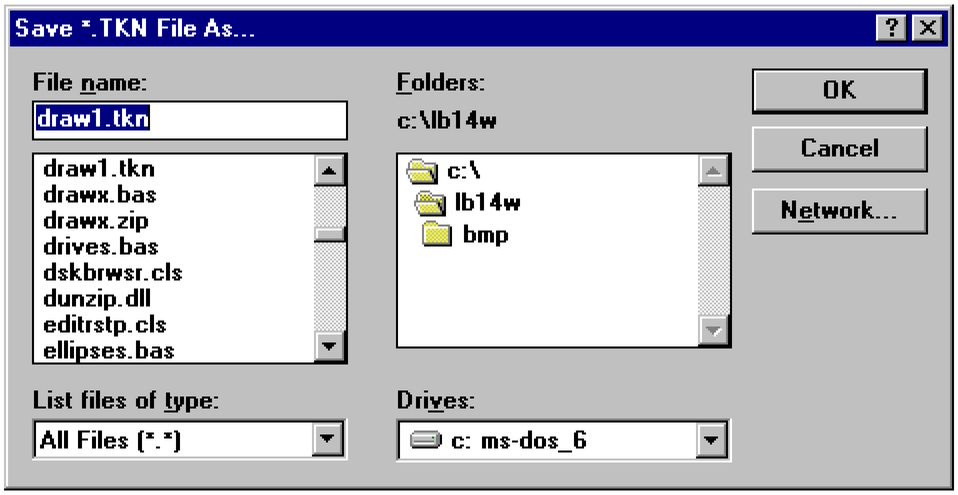


Now you will see small window indicating progress in creating our *.TKN file, like so:



Compiled 45 of 149 lines.

When the file is ready, you will be given a chance to enter a filename in place of the default (draw1.tkn in this case):



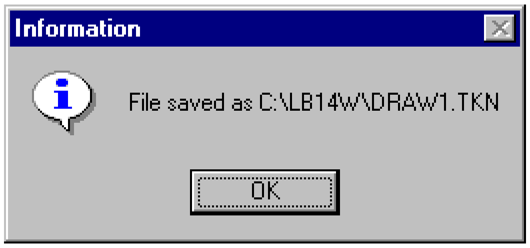
File name: draw1.tkn

Folders: c:\lb14w

List files of type: All Files (*.*)

Drives: c: ms-dos_6

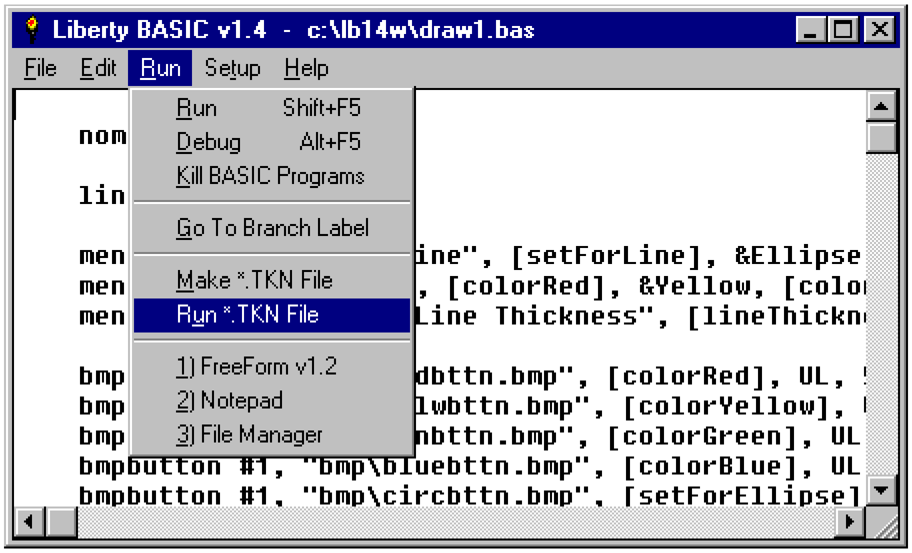
Finally, you will get a notice that we're all done:



File saved as C:\LB14W\DRAW1.TKN

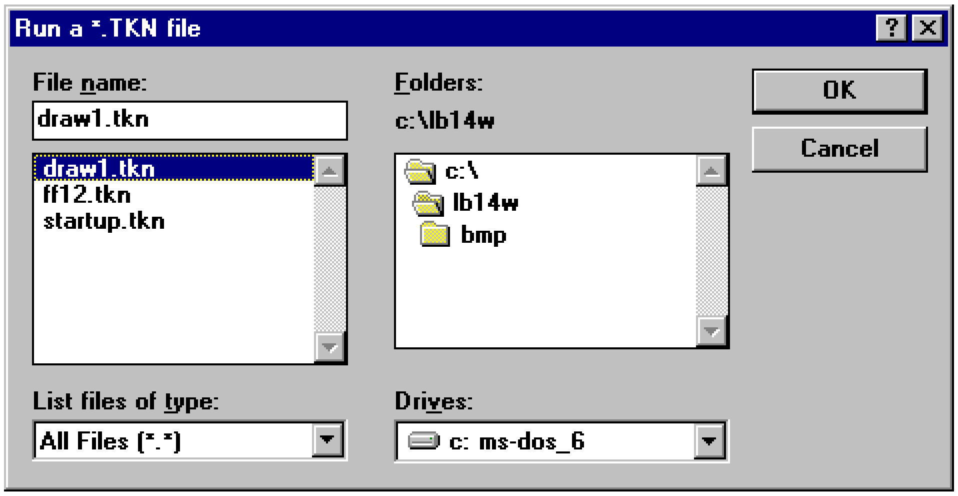
Clear the message by pressing Enter or by clicking on OK.

Now we are ready to run the .TKN file.   Pull down the Run menu and select Run *.TKN File as shown:

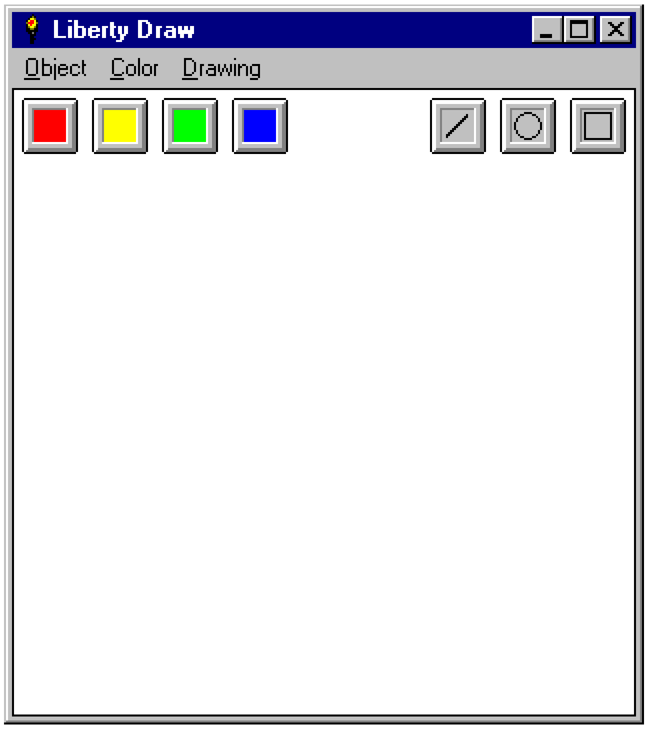


```
    nom
    lin
    men                              ine", [setForLine], &Ellipse
    men                              , [colorRed], &Yellow, [colo
    men                              Line Thickness", [lineThickn
    bmp                              dbttn.bmp", [colorRed], UL, 
    bmp                              lwbttn.bmp", [colorYellow], 
    bmp                              nbttn.bmp", [colorGreen], UL
    bmpbutton #1, "bmp\bluebttn.bmp", [colorBlue], UL
    bmpbutton #1, "bmp\circbttn.bmp", [setForEllipse]
```

A file dialog will be displayed containing a list of .TKN files.   Select the draw1.tkn file as shown and click on Ok.



Now the .TKN application will run, like so:


Liberty Draw
Object
Color
Drawing

Window Types

Liberty BASIC provides eighteen different kinds of window types, to which you can add as many controls as needed (see help section Controls - Menus, Buttons, Etc.). Here are their kinds and the commands associated with them:

The way that you would specify what kind of window to open would be as follows:

```
    open "Window Title" for type    as #handle
```

where type would be one of the eighteen descriptors (below).

Window types:

graphics	open a graphics window
graphics_fs	open a graphics window full screen (size of the screen)
graphics_nsb	open a graphics window w/no scroll bars
graphics_fs_nsb	open a graphics window full screen, w/no scroll bars

(View Graphics Window Commands)

text	open a text window
text_fs	open a text window full screen
text_nsb	open a text window w/no scroll bars
text_nsb_ins	open a text window w/no scroll bars, with inset editor

(View Text Window Commands)

spreadsheet	open a spreadsheet window

(Spreadsheet Window Commands)

window	open a basic window type
window_nf	open a basic window type without a sizing frame
dialog	open a dialog box
dialog_modal	open a modal dialog box
dialog_nf	open a dialog box without a frame
dialog_nf_modal	open a modal dialog box without a frame
dialog_fs	open a dialog box the size of the screen
dialog_nf_fs	open a dialog box without a frame the size of the screen

Controls and Events

When working with controls, you are asked to specify branch labels that are associated with user actions made on those controls (clicking, double-clicking, selecting, etc.). For example:

```
  button #main, "Accept", [userAccepts], UL, 10, 10
```

This adds a button to the window (#main) labeled "Accept". When the program is run, and the user clicks on this button, then execution branches to the routine at branch label [userAccepts]. This user clicking on the button generates an event. This is generally how branch label arguments are used in Liberty BASIC windows and controls.

Liberty BASIC can only respond to events when execution is halted at in INPUT statement. Look at this short program:

```
    ' This code demonstrates how to use checkboxes in your
    ' Liberty BASIC programs

    'nomainwin

    button #1, " &Ok ", [quit], UL, 120, 90
    checkbox #1.cb, "I am a checkbox", [set], [reset], 10, 10, 130, 20
    button #1, " Set ", [set], UL, 10, 50, 40, 25
    button #1, " Reset ", [reset], UL, 60, 50, 50, 25
    textbox #1.text, 10, 90, 100, 24

    WindowWidth = 190
    WindowHeight = 160
    open "Checkbox test" for dialog as #1
    print #1, "trapclose [quit]"

[waitHere]
    input r$

[set]

    print #1.cb, "set"
    goto [readCb]


[reset]

  print #1.cb, "reset"
    goto [readCb]

  end

[readCb]

    print #1.cb, "value?"
    input #1.cb, t$
    print #1.text, "I am "; t$
    goto [inputLoop]
```

```
[quit]

    close #1
    end
```

In the above code, Liberty BASIC opens a small window with a checkbox, a textbox, and a few buttons. After that, it stops at an INPUT statement just after the branch label [waitHere]. Now if the user clicks on this button or that, or in the checkbox, Liberty BASIC can handle the event and go to the appropriate branch label. If a user clicks on a button or causes some other event occur before we get back to our INPUT statement, the event is held for processing until we again reach that point in the program. Try copying the code above into your Liberty BASIC session and stepping through it with the debugger.

Controls - Menus, Buttons, Etc.

Here are the details for Liberty BASIC commands that add menus, buttons, listboxes, and more.

Controls and Events

Button
Menu
Listbox
Combobox
Textbox
Texteditor
Checkbox
Radiobutton
Groupbox
Statictext
Bmpbutton
Graphicsbox

Graphics Window Commands

Most of these commands work only with windows of type graphics and with the graphicbox control.

Here is an example using a graphics window:

```
    open "Drawing" for graphics as #handle

    print #handle, "home"
    print #handle, "down"
    print #handle, "fill cyan"
    print #handle, "north"
    print #handle, "circle 50"
    print #handle, "flush"

    input r$
```

And here is an example using a graphicbox:

```
    graphicbox #handle.gbox, 10, 10, 150, 150
    open "Drawing" for window as #handle

    print #handle.gbox, "home"
    print #handle.gbox, "down"
    print #handle.gbox, "fill cyan"
    print #handle.gbox, "north"
    print #handle.gbox, "circle 50"
    print #handle.gbox, "flush"

    input r$
```

Because graphics can involve many detailed drawing operations, Liberty BASIC does not force you to use just one print # statement for each drawing task. If you want to perform several operations you can use a single line for each as such:

```
    print #handle, "up"
    print #handle, "home"
    print #handle, "down"
    print #handle, "north"
    print #handle, "go 50"
```

Or if you prefer:

```
    print #handle, "up ; home ; down ; north ; go 50"
```

will work just as well, and executes slightly faster.

Here are the commands (in alphabetical order):

print #handle, "autoresize"

Graphicbox control only - This causes the **edges of the control to** maintain their distance from the edges of the overall window. If the user resizes the window, the texteditor control also resizes.

print #handle, "backcolor COLOR"

This command sets the color used when drawn figures are filled with a color. The same colors are available as with the color command above.

print #handle, "box x y"

Draw a box using the pen position as one corner, and x, y as the other corner.

print #handle, "boxfilled x y"

Draw a box using the pen position as one corner, and x, y as the other corner. Fill the box with the color specified using the command backcolor (see above).

print #handle, "circle r"

Draw a circle with radius r at the current pen position.

print #handle, "circlefilled r"

Draw a circle with radius r, and filled with the color specified using the command backcolor (see above).

print #handle, "cls"

Clear the graphics window to white, erasing all drawn elements

print #handle, "color COLOR"

Set the pen's color to be COLOR

Here is a list of valid colors (in alphabetical order):

black, blue, brown, cyan, darkblue, darkcyan, darkgray,
darkgreen, darkpink, darkred, green, lightgray, palegray,
pink, red, white, yellow

print #handle, "color red(0-255) green(0-255) blue(0-255)"

The second form of color lets you specify a pure RGB color. This only works with display modes greater than 256 colors. To create a violet color for example, mix red and blue like so:

```
print #handle, "color 127 0 127"
```

print #handle, "discard"

This causes all drawn items since the last flush to be discarded, but does not not force an immediate redraw, so the items that have been discarded will still be displayed until a redraw (see above).

print #handle, "down"

Just the opposite of up. This command reactivates the drawing process.

print #handle, "drawbmp bmpname x y"

This will draw a bitmap named bmpname (loaded beforehand with the LOADBMP statement, see command reference) at the location x y.

print #handle, "ellipse w h"

Draw an ellipse at the pen position of width w and height h.

print #handle, "ellipsefilled w h"

Draw an ellipse at the pen position of width w and height h. Fill the ellipse with the color specified using the command backcolor (see above).

print #handle, "delsegment n"

This causes the drawn segment identified as n to be removed from the window's list of drawn items. Then when the window is redrawn the deleted segment will not be included in the redraw.

print #handle, "fill COLOR"

or... print #handle, "color red(0-255) green(0-255) blue(0-255)"

Fill the window with COLOR. For a list of accepted colors see the color command below. The second form lets you specify a pure RGB color. This only works with

print #handle, "flush"

This ensures that drawn graphics 'stick'. Make sure to issue this command at the end of a drawing sequence to ensure that when the window is resized or overlapped and redrawn, its image will be retained. To each group of drawn items that is terminated with flush, there is assigned a segment ID number. See segment below.

print #handle, "font facename width height"

Set the pen's font to the specified face, width and height. If an exact match cannot be found, then Liberty BASIC will try to find a close match, with size being of more prominance than face.

Note that a font with more than one word in its name is specified by joining each word with an underscore _ character. For example, the font Times New Roman becomes Times_New_Roman, and the font Courier New becomes Courier_New.

Example:

```
print #handle, "font Times_New_Roman 8 15"
```

print #handle, "go D"

Go foreward D distance from the current position, and going in the current direction.

print #handle, "goto X Y"

Move the pen to position X Y. Draw if the pen is down.

print #handle, "getbmp bmpName x y width height"

Make a bitmap copied from the graphics window at x, y and using width and height.

print #handle, "home"

This command centers the pen in the graphics window.

print #handle, "line X1 Y1 X2 Y2"

Draw a line from point X1 Y1 to point X2 Y2. If the pen is up, then no line will be drawn, but the pen will be positioned at X2 Y2.

print #handle, "north"

Set the current direction to 270 (north). Zero degrees points to the right (east), 90 points down (south), and 180 points left (west).

print #handle, "pie w h angle1 angle2"

Draw a pie slice inside of an ellipse of width w and height h. Start the pie slice at angle1, and then sweep clockwise angle2 degrees if angle2 is positive, or sweep counter-clockwise angle2 degrees if angle2 is negative.

print #handle, "piefilled w h angle1 angle2"

Draw a pie slice inside of an ellipse of width w and height h. Start the slice at angle1, and then sweep clockwise angle2 degrees if angle2 is positive, or sweep counter-clockwise angle2 degrees if angle2 is negative. Fill the pie slice with the color specified using the command backcolor (see above).

print #handle, "place X Y"

Position the pen at X Y. Do not draw even if the pen is down.

print #handle, "posxy"

Return the position of the pen in x, y. This command must be followed by:

```
input #handle, xVar, yVar
```

which will assign the pen's position to xVar & yVar

print #handle, "print"

Send the plotted image to the Windows Print Manager for output.

print #handle, "redraw"

This will cause the window to redraw all flushed drawn segments. Any deleted segments will not be redrawn (see delsegment above). Any items drawn since the last flush will not be redrawn either, and will be lost.

print #handle, "rule rulename"

This command specifies whether drawing overwrites (rulename OVER) on the screen or uses the exclusive-OR technique (rulename XOR).

print #handle, "segment"

This causes the window to return the segment ID of the most recently flushed drawing segment. This segment ID can then be retrieved with an input #handle, varName and varName will contain the segment ID number. Segment ID numbers are useful for manipulating different parts of a drawing. For an example, see delsegment below.

print #handle, "setfocus"

This causes Windows to give input focus to this control. This means that if some other control in the same windows was highlighted and active, that this control now becomes the highlighted and active control, receiving keyboard input.

print #handle, "size S"

Set the size of the pen to S. The default is 1. This will affect the thickness of lines and figures plotted with most of the commands listed in this section.

print #handle, "\text"

Display text at the current pen position. Each additional \ in the text will cause a carraige return and line feed. Take for example, print #handle, "\text1\text2" will cause text1 to be printed at the pen position, and then text2 will be displayed directly under text1.

print #handle, "trapclose branchLabel"

This will tell Liberty BASIC to continue execution of the program at branchLabel if the user double clicks on the system menu box or pulls down the system menu and selects close (this command does not work with graphicsbox controls).

print #handle, "turn A"

Turn from the current direction using angle A and adding it to the current direction. A can be positive or negative.

print #handle, "up"

Lift the pen up. All go or goto commands will now only move the pen to its new position without drawing. Any other drawing commands will simply be ignored until the pen is put back down.

print #handle, "when event branchLabel"

This tells the window to process mouse events. These events occur when someone clicks, double-clicks, drags, or just moves the mouse inside of the graphics window. An event can also be the user pressing a key while the graphics window or graphicbox has the input focus (see the setfocus command, above). This provides a really simple mechanism for controlling flow of a program which uses the graphics window. For an example, see the program draw1.bas.

Sending print #handle, "when leftButtonDown [startDraw]" to any graphics window will tell that window to force a goto [startDraw] when the mouse points inside of that window and someone press the left mouse button down.

Whenever a mouse event does occur, Liberty BASIC places the x and y position of the mouse in the variables MouseX, and MouseY. The values will represent the number of pixels in x and y the mouse was from the upper left corner of the graphic window display pane.

If the expression print #handle, "when event" is used, then trapping
for that event is discontinued. It can however be reinstated at any time.

Events that can be trapped:

leftButtonDown	- the left mouse button is now down
leftButton Up	- the left mouse button has been released
leftButtonMove	- the mouse moved while the left button is down
leftButtonDouble	- the left button has been double-clicked
rightButtonDown	- the right mouse button is now down
rightButton Up	- the right mouse button has been released
rightButtonMove	- the mouse moved while the right button is down
rightButtonDouble	- the right button has been double-clicked
mouseMove	- the mouse moved when no button was down
characterInput	- a key was pressed while the graphics window has input focus (see the setfocus command, above)

Text Window Commands

The text window works a little differently. Whatever you print to a text window is displayed exactly as sent. The way to send commands to a text window is to make the ! character the first character in the string. It is also important to add a semicolon to the end of command line (a print #handle line with text window commands) as in the example below. If you don't, the print statement will force a carraige return into the text window each time you print a command to the window if you don't.

For example:

```
open "Example" for text as #1       'open a text window
print #1, "Hello World"             'print Hello World in the window
print #1, "!font helv 16 37" ;      'change the text window's font
print #1, "!line 1" ;               'read line 1
input #1, string$
print "The first line is:"
print string$
input "Press 'Return'"; r$
close #1                            'close the window
```

Note: Most of the commands listed below work with windows of type text and also with the texteditor control except where noted.

Here are the text window commands:

print #handle, "!autoresize";

Texteditor control only - This causes the **edges of the control to** maintain their distance from the edges of the overall window. If the user resizes the window, the texteditor control also resizes.

print #handle, "!cls" ;

Clears the text window of all text.

print #handle, "!contents varname$";
print #handle, "!contents #handle";

This has two forms as described above. The first form causes the contents of the text window to be replaced with the contents of varname$, and the second form causes the contents of the text window to be replaced with the contents of the stream referenced by #handle. This second form is useful for reading large text files quickly into the window.

Here is an example of the second form:

```
open "Contents of AUTOEXEC.BAT" for text as #aetext
open "C:\AUTOEXEC.BAT" for input as #autoexec
print #aetext, "!contents #autoexec";
```

```
    close #autoexec
    'stop here
    input a$
```

print #handle, "!contents?";

Returns the entire text of the window. After this command is issued, it must be followed by:

input #handle, string$

print #handle, "!copy" ;

This causes the currently selected text to be copied to the WINDOWS clipboard.

print #handle, "!cut" ;

This causes the currently selected text to be cut out of the text window and copied to the WINDOWS clipboard.

print #handle, "!font faceName width height" ;

Sets the font of the text window to the specified face of width and height. If an exact match cannot be found, then Liberty BASIC will try to match as closely as possible, with size figuring more prominently than face in the match.

Note that a font with more than one word in its name is specified by joining each word with an underscore _ character. For example, the font Times New Roman becomes Times_New_Roman, and the font Courier New becomes Courier_New.

Example:

```
print #handle, "!font Times_New_Roman 8 15";
```

print #handle, "!line #" ;

Returns the text at line #. If # is less than 1 or greater than the number of lines the text window contains, then "" (an empty string) is returned. After this command is issued, it must be followed by:

input #handle, string$

which will assign the line's text to string$

print #handle, "!lines" ;

Returns the number of lines in the text window. After this command is issued, it must be followed by:

input #handle, countVar

which will assign the line count to countVar

print #handle, "!modified?" ;

This returns a string (either "true" or "false") that indicates whether any data in the text window has been modified. This is useful for checking to see whether to save the contents of the window before closing it.

To read the result, an input #handle, varName$, must be performed after.

print #handle, "!origin?" ;

This causes the current text window origin to be returned. When a text window is first opened, the result would be row 1, column 1. To read the result an input #handle, rowVar, columnVar must be performed after.

print #handle, "!origin row column" ;

This forces the origin of the window to be row and column.

print #handle, "!paste" ;

This causes the text in the WINDOWS clipboard (if there is any) to be pasted into the text window at the current cursor position.

print #handle, "!selectall" ;

This causes everything in the text window to be selected.

print #handle, "!selection?" ;

This returns the highlighted text from the window. To read the result an input #handle, varName$ must be performed after.

print #handle, "!setfocus";

This causes Windows to give input focus to this control. This means that if some other control in the same windows was highlighted and active, that this control now becomes the highlighted and active control, receiving keyboard input.

print #handle, "!trapclose branchLabel" ;

This will tell Liberty BASIC to continue execution of the program at
branchLabel if the user double clicks on the system menu box
or pulls down the system menu and selects close (see ROLODEX1.BAS).

Spreadsheet Window Commands

The spreadsheet used in Liberty BASIC is composed of 35 rows of 26 columns labeled from A to Z. The upper-left-most cell is A1 and the lower-right-most cell is Z35. Each cell can contain one of three types of data: string, number, or formula. To enter one of these three types into any cell, simply move the selector over the cell on the spreadsheet and begin typing. When done entering that cell's contents, press 'Return'.

A string is entered by preceding it with an apostrophe '. Each cell is 11 characters wide so if the string is longer than 11 characters it will run into the next cell to its right.

A number is entered by entering its value, either an integer or a floating point number.

A formula is a simple arithmatic expression, using numbers (see above) or cell references. The result of the formula is displayed in the cell's position. Any arithmatic precedence is ignored, so any formula is always evaluated from left to right and parenthesis are not permitted (They aren't needed).

A formula to compute the average of 3 cells might be: a1 + a2 + a3 / 3

The spreadsheet is a very special widget. Alone it is a very simple but complete spreadsheet. But being able to send it commands and data and to be able to read back data from it via Liberty BASIC makes it a very powerful tool. For examples, see grapher.bas and customer.bas.

Modes: The spreadsheet has two modes, manual and indirect. Manual mode means that that the operator can freely move about from cell to cell with the arrow keys. He/she can also insert formulas in manual mode. Using indirect mode, the user can only move to cells predefined by the controlling application, which also decides what type of data is contained by each cell, either string or number.

Here are the commands:

```
print #handle, "manual"
```

The manual mode is the default setting. This mode permits the user to move the cell selector wherever he/she wants and to enter any of three data types into any cell: number, string, formula

```
print #handle, "format COLUMN right|fixed|none"
```

This command lets the application control formatting for an individual column (COLUMN can be any letter A .. Z).

right - right justify column
fixed - assume 2 decimal places for numbers, and right justify also
none - left justify, default

```
print #handle, "indirect"
```

The indirect mode is the most useful when using a spreadsheet for data entry. It enables the application to control which cells the user has access to, and what kind of information they can contain.

```
print #handle, "cell ADDRESS CONTENTS"
```

Place CONTENTS into the cell at ADDRESS. ADDRESS can be any cell address from A1 to Z35. The letter A to Z must be in uppercase. CONTENTS can be any valid string, number or formula (see above).

```
print #handle, "user ADDRESS string|number"␡
```

Set aside the cell at ADDRESS (same rules apply as for ADDRESS in command cell, above) as a user cell and specify the data it contains to be either a string or a number (data entered will be automatically converted to correct type). This command is only effective when indirect mode is in effect (see above).

```
print #handle, "select ADDRESS"
```

Place the selector over the cell at ADDRESS (again, same rules). It is important to place the selector over the first cell that the user will edit.

```
print #handle, "result? ADDRESS"
```

Answer the result or value of the cell at ADDRESS (again, same rules). If ADDRESS is not a valid cell address, then an empty string will be returned. This command must be followed by:

```
input #handle, var$   (or   input #handle, var   if number expected)
```

which will leave the desired cell's contents in var$ (or var)

```
print #handle, "formula? ADDRESS"
```

Answer the formula of the cell at ADDRESS (again, same rules). This command must also be followed with:

```
input #handle, var$   (should always be a string returned)
```

which will leave the desired cell's formula in var$

```
print #handle, "flush"
```

This commands forces the spreadsheet to display its most up to date results.

```
print #handle, "load pathFileName"
```

This causes a Liberty BASIC spreadsheet file (which always have an .abc extension) named pathFileName to be loaded, replacing the current data set.

print #handle, "save pathFileName"

This causes spreadsheet data set (which will always have an .abc extension) to be saved to disk at pathFileName.

print #handle, "modified?"

This returns a string (either "true" or "false") that indicates whether any data in the spreadsheet has been modified.   This is useful for checking to see whether to save the contents of the window before closing it.

To read the result, an input #handle, varName$, must be performed after.

print #handle, "nolabels"

This turns off the row and column labels.

print #handle, "labels"

This turns on the row and column labels.

print #handle, "trapclose branchLabel"

This will tell Liberty BASIC to continue execution of the program at branchLabel if the user double clicks on the system menu box or pulls down the system menu and selects close (see rolodex1.bas).

Button

Buttons are easily added to Liberty BASIC windows. The format is simple:

```
BUTTON #handle.ext, "Label", [branchLabel], corner, distX, distY
open "A Window!" for graphics as #handle
```

or

```
BUTTON #handle.ext, "Label", [branchLabel], corner, x, y, width, height
open "A Window!" for graphics as #handle
```

By placing at least one button statement before the open statement, we can add button(s) to the window. Let's examine each part of the button statement:

#handle.ext - This needs to be the same as the handle of the window, with an optional unique extension.

"Label" - This is the text displayed on the button. If only one word is used, then the quotes are optional.

[branchLabel] - This controls what the button does. When the user clicks on the button, then program execution continues at [branchLabel] as if the program had encountered a goto [branchLabel] statement.

corner, distX, distY - Corner is used to indicate which corner of the window to anchor the button to. DistX and distY specify how far from that corner in x and y to place the button. The following values are permitted for corner:

UL - Upper Left Corner
UR - Upper Right Corner
LL - Lower Left Corner
LR - Lower Right Corner

width, height - These are optional. If you do not specify a width and height, then Liberty BASIC will automatically determine the size of the button.

Menu

Menus are easily added to Liberty BASIC windows.   The format is simple:

```
    MENU #handle, "Title", "Line1", [branchLabel1], "Line2", [branchLabel2], ...
    open "A Window!" for graphics as #handle
```

By placing at least one menu statement before   the open statement, we can add menu(s) to the window.   Let's examine each part of the menu statement:

#handle - This needs to be the same as the handle of the window.

"Title" - This is the title displayed on the menu bar.   If only one word is used, then the quotes are optional.   By including an ampersand and in front of the character desired, you can turn that character into a hot-key.   For example, if the title is "&Help", the title will appear as Help.

"Line1" and [branchLabel1] - This is a line item seen when the menu is pulled down.   [branchLabel1] is the place where execution continues if this menu item is selected.   Like "Title", "Line1" requires quotes only if there is more than one word.   The ampersand & character is used to assign a hot-key to the label, as in "Title", above.

"Line2" and [branchLabel2] - This is a second line item and branch label for the menu.   You can have as many is needed, going on with "Line3 . . . 4 . . .   5", etc.

Adding seperators between menu items to group them is easy.   Simply add a bar | character between each group of items.   For example:

```
    . . . "&Red", [colorRed], |, "&Size", [changeSize] . . .
```

adds a line seperator between the Red and Size menu items.

Listbox

Listboxes in Liberty BASIC can be added to any windows that are of type graphics, window, and dialog. They provide a list selection capability to your Liberty BASIC programs. You can control the contents, position, and size of the listbox, as well as where to transfer execution when an item is selected. The listbox is loaded with a collection of strings from a specified string array, and a reload command updates the contents of the listbox from the array when your program code changes the array.

Here is the syntax:

```
  LISTBOX #handle.ext, array$(, [branchLabel], xPos, yPos, wide, high
```

#handle.ext - The #handle part of this item needs to be the same as the handle of the window you are adding the listbox to. The .ext part needs to be unique so that you can send commands to the listbox and get information from it later.

array$(- This is the name of the array (must be a string array) that contains the contents of the listbox. Be sure to load the array with strings before you open the window. If some time later you decide to change the contents of the listbox, simply change the contents of the array and send a reload command.

[branchLabel] - This is the branch label where execution begins when the user selects an item from the listbox by double-clicking. Selection by only single clicking does not cause branching to occur.

xPos & yPos - This is the distance in x and y (in pixels) of the listbox from the upper-left corner of the window.

wide & high - This determines just how wide and high (in pixels) the listbox is.

Here are the commands for listbox:

```
  print #handle.ext, "select string"
```

Select the item the same as string and update the display.

```
  print #handle.ext, "selectindex i"
```

Select the item at index position i and update the display.

```
  print #handle.ext, "selection?"
```

Return the selected item. This must be followed by the statement:

```
    input #handle.ext, selected$
```

This will place the selected string into selected$. If there is no selected item, then selected$ will be a string of zero length (a null string).

print #handle.ext, "selectionindex?"

Return the index of the selected item.   This must be followed by the statement:

```
input #handle.ext, index
```

This will place the index of the selected string into index.   If there is no selected item, then index will be set to 0.

print #handle.ext, "reload"

This will reload the listbox with the current contents of its array and will update the display.

print #handle.ext, "font facename width height"

This will set the listbox's font to the font specified.   Windows' font selection algorithm is designed to make an approximate match if it cannot figure out exactly which font you want.

print #handle.ext, "singleclickselect"

This tells Liberty BASIC to jump to the control's branch label on a single click, instead of the default double click.

print #handle.ext, "setfocus"

This will cause the listbox to receive the input focus.   This means that any keypresses will be directed to the listbox.

```
    ' Sample program.   Pick a contact status

        options$(0) = "Cold Contact Phone Call"
        options$(1) = "Send Literature"
        options$(2) = "Follow Up Call"
        options$(3) = "Send Promotional"
        options$(4) = "Final Call"

        listbox #status.list, options$(, [selectionMade], 5, 35, 250, 90
        button #status, Continue, [selectionMade], UL, 5, 5
        button #status, Cancel, [cancelStatusSelection], UR, 15, 5
        WindowWidth = 270 : WindowHeight = 180
        open "Select a contact status" for window as #status

    input r$

[selectionMade]
    print #status.list, "selection?"
```

```
        input #status.list, selection$
        notice selection$ + " was chosen"
        close #status
        end

    [cancelStatusSelection]
        notice "Status selection cancelled"
        close #status
        end
```

Control of the listbox in the sample program above is provided by printing commands to the listbox, just as with general window types in Liberty BASIC. We gave the listbox the handle #status.list, so to find out what was selected, we use the statement print #status.list, "selection?". Then we must perform an input, so we use input #status.list, selection$, and the selected item is placed into selection$. If the result is a string of length zero (a null string), this means that there is no item selected.

Combobox

Comboboxes are a lot like listboxes, but they are designed to save space. Instead of showing an entire list of items, they show only the selected one. If you don't like the selection, then you click on its button (to the right), and a list appears. Then you can browse the possible selections, and pick one if so desired. When the selection is made, the new selection is displayed in place of the old. If you don't want to make a new selection, just click on the combobox's button again, and the list will disappear.

Comboboxes in Liberty BASIC can be added to any windows that are of type graphics, window, and dialog. They provide a list selection capability to your Liberty BASIC programs. You can control the contents, position, and size of the combobox, as well as where to transfer execution when an item is selected. The combobox is loaded with a collection of strings from a specified string array, and a reload command updates the contents of the combobox from the array when your program code changes the array.

Here is the syntax:

```
  COMBOBOX #handle.ext, array$(, [branchLabel], xPos, yPos, wide, high
```

#handle.ext - The #handle part of this item needs to be the same as the handle of the window you are adding the listbox to. The .ext part needs to be unique so that you can send commands to the listbox and get information from it later.

array$(- This is the name of the array (must be a string array) that contains the contents of the listbox. Be sure to load the array with strings before you open the window. If some time later you decide to change the contents of the listbox, simply change the contents of the array and send a reload command.

[branchLabel] - This is the branch label where execution begins when the user selects an item from the listbox by double-clicking. Selection by only single clicking does not cause branching to occur.

xPos & yPos - This is the distance in x and y (in pixels) of the listbox from the upper-left corner of the window.

wide & high - This determines just how wide and high (in pixels) the listbox is. Height refers to how far down the selection list reaches when the combobox's button is clicked, not to the size of the initial selection window.

Here are the commands for combobox:

```
  print #handle.ext, "select string"
```

Select the item the same as string and update the display.

```
print #handle.ext, "selectindex i"
```

Select the item at index position i and update the display.

```
print #handle.ext, "selection?"
```

Return the selected item. This must be followed by the statement:

```
input #handle.ext, selected$
```

This will place the selected string into selected$. If there is no selected item, then selected$ will be a string of zero length (a null string).

```
print #handle.ext, "selectionindex?"
```

Return the index of the selected item. This must be followed by the statement:

```
input #handle.ext, index
```

This will place the index of the selected string into index. If there is no selected item, then index will be set to 0.

```
print #handle.ext, "reload"
```

This will reload the combobox with the current contents of its array and will update the display.

```
print #handle.ext, "setfocus"
```

This will cause the combobox to receive the input focus. This means that any keypresses will be directed to the combobox.

For a sample program, see the included file dialog3.bas.

Textbox

The textbox command lets you add a single item, single line text entry/editor box to your windows. It is useful for generating forms in particular.

The syntax for textbox is simply:

```
  TEXTBOX #handle.ext, xpos, ypos, wide, high
```

#handle.ext - The #handle part must be the same as for the window you are adding the textbox to. The .ext part must be unique for the textbox.

xpos & ypos - This is the position of the textbox in x and y from the upper-left corner of the window.

wide & high - This is the width and height of the textbox in pixels.

Textbox understands these commands:

```
    print #handle.ext, "a string"
```

This sets the contents of the textbox to be "a string".

```
    print #handle.ext, "!contents?"
```

This fetches the contents of the textbox. This must be followed by:

```
    input #handle.ext, varName$
```

The contents will be placed into varName$

```
  print #handle.ext, "setfocus"
```

This will cause the textbox to receive the input focus. This means that any keypresses will be directed to the textbox.

```
  ' sample program

    textbox #name.txt, 20, 10, 260, 25
    button #name, "OK", [titleGraph], LR, 5, 0
    WindowWidth = 350 : WindowHeight = 90
    open "What do you want to name this graph?" for window_nf as #name
    print #name.txt, "untitled"

[mainLoop]
    input wait$

[titleGraph]
    print #name.txt, "!contents?"
```

```
input #name.txt, graphTitle$
notice "The title for your graph is: "; graphTitle$
close #name
end
```

Texteditor

Texteditor is a control that like Textbox, but with scroll bars, and with an enhanced command set. The commands are essentially the same as that of a window of type text. NOTICE that all of these commands start with an exclamation point, because the control will simple display anything printed to it if it doesn't start with an exclamation point.

Here is the syntax for texteditor:

```
TEXTEDITOR #handle.ext, xpos, ypos, wide, high
```

Here are the texteditor commands:

```
print #handle, "!cls" ;
```

Clears the text window of all text.

```
print #handle, "!font faceName width height" ;
```

Sets the font of the text window to the specified face of width and height. If an exact match cannot be found, then Liberty BASIC will try to match as closely as possible, with size figuring more prominently than face in the match.

```
print #handle, "!line #" ;
```

Returns the text at line #. If # is less than 1 or greater than the number of lines the text window contains, then "" (an empty string) is returned. After this command is issued, it must be followed by:

```
input #handle, string$
```

which will assign the line's text to string$

```
print #handle, "!lines" ;
```

Returns the number of lines in the text window. After this command is issued, it must be followed by:

```
input #handle, countVar
```

which will assign the line count to countVar

```
print #handle, "!modified?" ;
```

This returns a string (either "true" or "false") that indicates whether any data in the text window has been modified. This is useful for checking to see whether to save the contents of the window before closing it.

To read the result, an input #handle, varName$, must be performed after.

print #handle, "!selection?" ;

This returns the highlighted text from the window. To read the result an input #handle, varName$ must be performed after.

print #handle, "!selectall" ;

This causes everything in the text window to be selected.

print #handle, "!copy" ;

This causes the currently selected text to be copied to the WINDOWS clipboard.

print #handle, "!cut" ;

This causes the currently selected text to be cut out of the text window and copied to the WINDOWS clipboard.

print #handle, "!paste" ;

This causes the text in the WINDOWS clipboard (if there is any) to be pasted into the text window at the current cursor position.

print #handle, "!origin?" ;

This causes the current text window origin to be returned. When a text window is first opened, the result would be row 1, column 1. To read the result an input #handle, rowVar, columnVar must be performed after.

print #handle, "!origin row column" ;

This forces the origin of the window to be row and column.

Checkbox

Adds a checkbox control to the window referenced by #handle. Checkboxes have two states, set and reset. They are useful for getting input of on/off type information.

The syntax of this command is:

```
  CHECKBOX #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height
```

Here is a description of the parameters of the CHECKBOX statement:

#handle	- This must be the same as the #handle of the window you are adding the statictext to. If #handle.ext is used, this allows your program to print commands to the checkbox control.
"label"	- This contains the visible text of the checkbox
[set]	- This is the branch label to goto when the user sets checkbox by clicking on it.
[reset]	- This is the branch label to goto when the user resets the checkbox by clicking on it.
xOrigin	- This is the x position of the checkbox relative to the upper left corner of the window it belongs to.
yOrigin	- This is the y position of the checkbox relative to the upper left corner of the window it belongs to.
width	- This is the width of the checkbox control
height	- This is the height of the checkbox control

Checkboxes understand these commands:

```
print #handle.ext, "set"
```

This sets the checkbox.

```
print #handle.ext, "reset"
```

This resets the checkbox.

```
print #handle.ext, "value?"
```

This returns the status of the checkbox. Follow this statement with:

```
input #handle.ext, result$
```

The variable result$ will be either "set" or "reset".

print #handle.ext, "setfocus"

This will cause the combobox to receive the input focus. This means
that any keypresses will be directed to the combobox.

Usage:

See the included program checkbox.bas for an example of how to use checkboxes.

Radiobutton

Adds a radiobutton control to the window referenced by #handle.   Radiobuttons have two states, set and reset.   They are useful for getting input of on/off   type information.

The syntax of this command is:

```
  RADIOBUTTON #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height
```

All radiobuttons on a given window are linked together, so that if you set one by clicking on it, all the others will be reset.

Here is a description of the parameters of the RADIOBUTTON statement:

#handle	- This must be the same as the #handle of the window you are adding the statictext to. If #handle.ext is used, this allows your program to print commands to the statictext control (otherwise all you'll be able to do is set it and forget it).
"label"	- This contains the visible text of the radiobutton
[set]	- This is the branch label to goto when the user sets the radiobutton by clicking on it.
[reset]	- This is the branch label to goto when the user resets the radiobutton by clicking on it. (this doesn't actually do anything because radiobuttons can't be reset by clicking on them).
xOrigin	- This is the x position of the radiobutton relative to the upper left corner of the window it belongs to.
yOrigin	- This is the y position of the radiobutton relative to the upper left corner of the window it belongs to.
width	- This is the width of the radiobutton control
height	- This is the height of the radiobutton control

Radiobuttons understand these commands:

```
print #handle.ext, "set"
```

This sets the radiobutton.

```
print #handle.ext, "reset"
```

This resets the radiobutton.

```
print #handle.ext, "value?"
```

This returns the status of the radiobutton. Follow this statement with:

```
input #handle.ext, result$
```

The variable result$ will be either "set" or "reset".

```
print #handle.ext, "setfocus"
```

This will cause the radiobutton to receive the input focus. This means that any keypresses will be directed to the radiobutton.

Usage:

See the included program radiobtn.bas for an example of how to use radiobuttons.

Groupbox

Like statictext, groupbox lets you place instructions or labels into your windows. But groupbox also draws a box that can be used to group related dialog box components.

The syntax of this command is:

```
GROUPBOX #handle, "string", xpos, ypos, wide, high
```

#handle - This must be the same as the #handle of the window you are adding the groupbox to.

"string" - This is the text component of the groupbox.

xpos & ypos - This is the distance of the groupbox in x and y (in pixels) from the upper-left corner of the screen.

wide & high - This is the width and height of the groupbox.

For an example of how groupbox is used, see the included source file dialog3.bas.

NOTE: Groupboxes are not really intended for use in windows of type graphics, and may not work properly in that context.

Statictext

Statictext lets you place instructions or labels into your windows.   This is most often used with a textbox to describe what to type into it.

The syntax of this command is:

```
  STATICTEXT #handle, "string", xpos, ypos, wide, high

    or

  STATICTEXT #handle.ext, "string", xpos, ypos, wide, high
```

#handle - This must be the same as the #handle of the window you are adding the statictext to. If #handle.ext is used, this allows your program to print commands to the statictext control (otherwise all you'll be able to do is set it and forget it).

"string" - This is the text component of the statictext.

xpos & ypos - This is the distance of the statictext in x and y (in pixels) from the upper-left corner of the screen.

wide & high - This is the width and height of the statictext. You must specify enough width and height to accomodate the text in "string".

Statictext understands only this command:

```
    print #handle.ext, "a string"
```

This sets the contents (the visible label) of the statictext to be "a string".   The handle must be of form #handle.ext so that you can print to the control.

```
    'sample program

    statictext #member, "Name", 10, 10, 40, 18
    statictext #member, "Address", 10, 40, 70, 18
    statictext #member, "City", 10, 70, 60, 18
    statictext #member, "State", 10, 100, 50, 18
    statictext #member, "Zip", 10, 130, 30, 18

    textbox #member.name, 90, 10, 180, 25
    textbox #member.address, 90, 40, 180, 25
    textbox #member.city, 90, 70, 180, 25
    textbox #member.state, 90, 100, 30, 25
    textbox #member.zip, 90, 130, 100, 25

    button #member, "&OK", [memberOK], UL, 10, 160

    WindowWidth = 300 : WindowHeight = 230
    open "Enter Member Info" for dialog as #member
    input r$

[memberOK]
    print #member.name, "!contents?" : input #member.name, name$
    print #member.address, "!contents?" : input #member.address, address$
    print #member.city, "!contents?" : input #member.city, city$
    print #member.state, "!contents?" : input #member.state, state$
    print #member.zip, "!contents?" : input #member.zip, zip$
    cr$ = chr$(13)
    note$ = name$ + cr$ + address$ + cr$ + city$ + cr$ + state$ + cr$ + zip$
    notice "Member Info" + cr$ + note$

    close #member
    end
```

Bmpbutton

Bmpbuttons are easily added to Liberty BASIC windows. The format is simple:

```
BMPBUTTON #handle.ext, "filename.bmp", [branchLabel], corner, distX, distY
open "A Window!" for graphics as #handle
```

By placing at least one bmpbutton statement before the open statement, we can add bmpbutton(s) to the window. Let's examine each part of the button statement:

#handle.ext - This needs to be the same as the handle of the window, with an optional unique extension.

"Label" - This is the text displayed on the button. If only one word is used, then the quotes are optional.

[branchLabel] - This controls what the button does. When the user clicks on the button, then program execution continues at [branchLabel] as if the program had encountered a goto [branchLabel] statement.

corner, distX, distY - Corner is used to indicate which corner of the window to anchor the button to. DistX and distY specify how far from that corner in x and y to place the button. The following values are permitted for corner:

UL - Upper Left Corner
UR - Upper Right Corner
LL - Lower Left Corner
LR - Lower Right Corner

Graphicsbox

The graphicsbox is a control that can be added to any window type and that understands all of the commands of a graphics window type.

The syntax for this command is:

```
    graphicsbox #handle.ext, xOrg, yOrg, width, height
```

#handle.ext - The #handle part must be the same as for the window you are adding the graphicsbox to. The .ext part must be unique for the graphicsbox.

xOrg & yOrg - This is the position of the textbox in x and y from the upper-left corner of the window.

width & height - This is the width and height of the textbox in pixels.

Jump to commands for graphicsbox

Trapping the close event

It is useful for Liberty BASIC program windows to trap the close event. This keeps the windows from closing, and directs program flow to a branch label that your program specifies. At that place in your program you can decide to ask for verification that the window should be closed, and/or perform some sort of cleanup (closing files, writing ini data, etc.).

The trapclose command works with all window types.

Here is the format for trapclose:

```
  print #handle, "trapclose branchLabel"
```

This will tell Liberty BASIC to continue execution of the program at branchLabel if the user double clicks on the system menu box or pulls down the system menu and selects close (see buttons1.bas example below).

The trapclose code in buttons1.bas looks like this:

```
    open "This is a turtle graphics window!" for graphics_nsb as #1
    print #1, "trapclose [quit]"

[loop]      ' stop and wait for buttons to be pressed
    input a$
    goto [loop]
```

And then the code that is executed when the window is closed looks like this:

```
[quit]
    confirm "Do you want to quit Buttons?"; quit$
    if quit$ = "no" then [loop]
    close #1
    end
```

Since this only works when the program is halted at an input statement, the special variable TrapClose permits detection of the window close when you are running a continuous loop that doesn't stop to get user input. As long as TrapClose <> "true", then the window has not been closed. Once it has been determined that TrapClose = "true", then it must be reset to "false" via the BASIC LET statement. See clock.bas for an example.

Making API and DLL Calls

Liberty BASIC v1.3 adds the ability to make 16-bit Windows API calls and to bind to third party Dynamic-Link-Libraries.   You now have access to hundreds of functions provided in Windows and from third-party sources that can greatly increase your productivity.

Notes to experienced Windows programmers:
 - Liberty BASIC processes Windows messages with its own handler.
 - Callbacks are not supported in this version of Liberty BASIC.

Informational resources about APIs/DLLs

This help file is designed as a basic introduction on using Windows APIs and DLLs in Liberty Basic. For a detailed understanding of how and why to call APIs and DLLs, you will need to study a separate volume designed to explain Windows programming in detail.

Microsoft includes detailed references with its C compilers, but it is usually good to supplement these with other books. Here are some suggestions:

For a general understanding of Windows 3.1 (16 bit Windows) programming:

> Programming Windows 3.1
> by Charles Petzold
> Microsoft Press
> ISBN 1-55615-395-4

For a catalog of Windows API calls and their function:

> Windows API Bible
> by James L. Conger
> Waite Group Press
> ISBN 1-878739-15-8

What are APIs/DLLs?

API stands for Application Programming Interface. Windows APIs are the function calls that are the fundamental building blocks of Windows programming. Although the term API actually refers to the complete set of function calls, the term API is also often applied to just a single defined function call of the entire API. So it is often said, "I made an API call," or, "I want my program to call an API that does such and such..."

Any time you use any Windows program at all, and in fact each and every time you load Windows, many API calls are made. There are API calls that manage memory, create and destroy windows, read keyboard and mouse actions, draw graphics, and much more. Liberty BASIC makes many of these API calls for you behind the scenes when you write your own Liberty BASIC programs.

However, in the context of the BASIC language, it would be a tall order (and largely unnecessary) to create BASIC-like statements and functions to implement each and every Windows API call. So for those who already have a working knowledge and for those willing to exert themselves to learn about such things, we have implemented a way to call most Windows APIs from Liberty BASIC.

A complete reference of the Windows API set is not included with this copy of Liberty BASIC. To supply this documentation would require the inclusion of a very thick book. Some example programs (see CALLDLL?.BAS) are included.

What are DLLs?

DLL stands for Dynamic-Link-Library. A DLL is a file containing executable code, like an EXE or COM file.
Instead of containing a complete program, a DLL contains functions that can be used by other running programs. These functions might contain code to provide services not built into Windows, for example the ability to perform some kind of data compression. A Windows program would use these functions after it begins executing. It does this by opening the DLL file and calling functions from it. The programmer must already know what these functions are when the program's code is written.

Each DLL has its own Application Programming Interface (again API) that specifies how to make calls to its functions, and in fact, Windows APIs are functions within DLLs that are supplied with Windows. When calling an API from within Liberty BASIC it is necessary to open the appropriate DLL before making the call.

How to make API/DLL calls

Since making an API call is really the same as making a DLL call, the following applies to both. In general, calling an API/DLL function works like so:

1) Open the desired DLL
2) Call the function or functions
3) Close the DLL

You might decide to open the desired DLL(s) when your program starts and close them when program execution is completed, or you might decide to open the DLL(s) just before calling the functions and then close them immediately afterward. The choice is yours.

The following statements/functions are available for making API/DLL calls:

```
OPEN "filename.dll" for dll as #handle
```

- This form of OPEN opens a desired DLL so that your program's code can call functions in it.

```
CALLDLL #handle, "function", parm1 as type1[, parm2 as type2, ... ], result
as returnType
```

- This statement calls a named function in an OPENed DLL

```
HWND(#handle)
```

- This function returns a window handle for the Liberty BASIC #handle

```
STRUCT name, field1 as type1 [, field2 as type2, ... ]
```

The STRUCT statement builds a specified structure that might be required when making some kinds of API/DLL calls. Below see an example that shows how to build a rectangle structure often used in making Windows API calls.

Using types with STRUCT and CALLDLL

Here is a short code sample (but not a complete program) using all of the above statements/functions:

```
    ' create the structure winRect
    struct winRect, _
        orgX as ushort, _
        orgY as ushort, _
        extentX as ushort, _
        extentY as ushort

    'open USER.DLL
    open "user" for dll as #user

    'get the window handle for #main (a standard Liberty BASIC window)
    hMain = hwnd(#main)

    'call the GetWindowRect API to load the window position/size into
winRect
    calldll #user, "GetWindowRect", _
        hMain as word, _
        winRect as struct, _
        result as void

    'extract the position information out of our struct
    xOrg = winRect.orgX.struct
    yOrg = winRect.orgY.struct

    'call the MoveWindow API to resize the window to a predefined size
    calldll #user, "MoveWindow", _
        hMain as word, _
        xOrg as short, _
        yOrg as short, _
        openingWidth as short, _
        expandedHeight as short, _
        1 as word, _
        result as void

    'get the window handle to a button in our Liberty BASIC window
    hndl = hwnd(#main.showMore)

    'call the ShowWindow API, passing _SW_HIDE to hide the button
    calldll #user, "ShowWindow", _
        hndl as word, _
        _SW_HIDE as ushort, _
        result as word

    'close USER.DLL
    close #user
```

In the above example, we call three APIs from USER.DLL, which is a standard Windows library.

Here's what the code does:

1) create the structure winRect
2) open USER.DLL
3) get the window handle for #main (a standard Liberty BASIC window)
4) call the GetWindowRect API to load the window position/size into winRect
5) extract the position information out of our struct
6) call the MoveWindow API to resize the window to a predefined size
7) get the window handle to a button in our Liberty BASIC window
8) call the ShowWindow API, passing _SW_HIDE to hide the button
9) close USER.DLL

Constants

Liberty BASIC has a library of defined constants (_SW_HIDE being one). This is equivalent to the definitions made in the windows.h file that comes with most C compilers. The way to inline a Windows constant is to take the name of the constant and place an underscore in front of it. _SW_HIDE is the same as the Windows constant SW_HIDE.

If you try to use a constant that is not defined inside of Liberty BASIC, you will get an undefined constant message when you try to run the program. In this case, you should try to determine the numeric value for that constant and use it in your program. If you discover a Windows constant that is not defined in Liberty BASIC, please report it to us so that we can add it to the next release.

Example Programs

For some examples showing how to call APIs, examine the programs named CALLDLL?.BAS that are included with your copy of Liberty BASIC.

Caveats

There are certain things to be aware of when making API calls and when calling external functions from a DLL including:

- Take care when calling APIs and DLLs because it is possible to cause Windows or Liberty BASIC to become unstable or crash if API and/or DLL function calls are called incorrectly, out of order, or with incorrect parameter information.

- Shoptalk Systems does not provide technical support for the Windows API or for third party DLLs that you may purchase or otherwise obtain. We are happy to help you with any questions about Liberty BASIC commands and features that are related to calling APIs and DLLs, but we reserve the right to not answer questions about other products (including Windows).

Using the Runtime Engine

Notice: This part of the help file describes a feature of the registered version of Liberty BASIC. This unregistered shareware version of Liberty BASIC does not include the RUN.EXE runtime engine.

The RUN.EXE runtime engine will allow you to create standalone programs from your Liberty BASIC *.TKN files. To use the runtime engine, you must place a PASSWORD statement before the first SCAN or INPUT statement in your program. Your password is based on your name. Both are provided to you on a printed sheet included with the Liberty BASIC package. The name and password must be spelled exactly as presented on the included sheet and they must be in quotes.

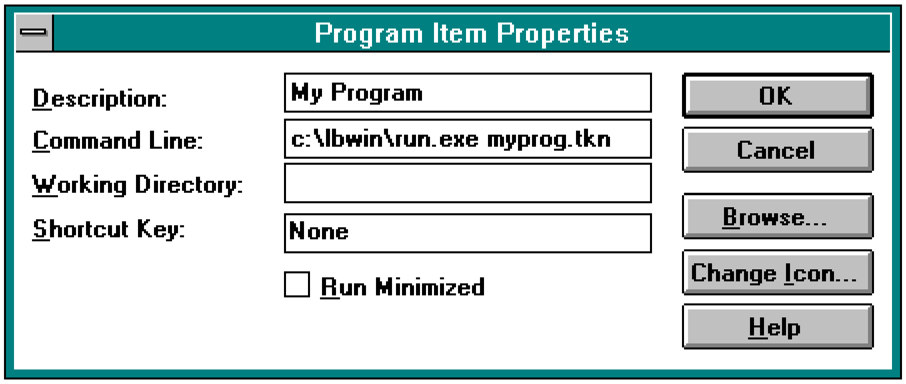
```
'Here is an example
password "Your Name", "?password?"
```

Using RUN.EXE

First make a *.TKN file from your *.BAS file (see Overview of Liberty BASIC).

Windows 3.1

Create a new Program Manager item, like so:



Program Item Properties

Description: My Program

Command Line: c:\lbwin\run.exe myprog.tkn

Working Directory:

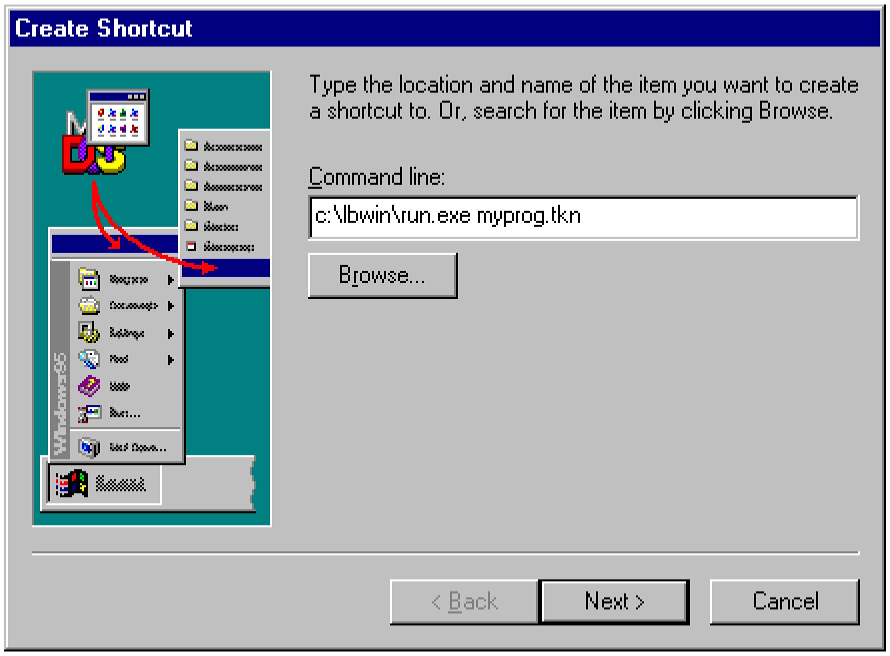
Shortcut Key: None

☐ Run Minimized

This will create an icon from the Program Manager that you can double-click on to start your program. Make sure that when your program closes all its windows to quit, that it terminates properly with an END statement.

Windows 95

To do this in Windows 95 you need to create a shortcut that will run the program (see your Windows 95 documentation for more on shortcuts). This is how it might look:



Type the location and name of the item you want to create a shortcut to. Or, search for the item by clicking Browse.

Command line:

c:\lbwin\run.exe myprog.tkn

Preparing for distribution

You can share or sell programs that you write in Liberty BASIC.   No fee or royalty payment is necessary.
The only requirements are:

a) That you limit the files that you distribute to the list below:

```
RUN.EXE
VWVM11.DLL
VWBAS11.DLL
VWSIGNON.DLL
VWSIGNON.BMP (replace this with your own 16 color bmp)
VWFONT.DLL
VWFLOAT.DLL
VWABORT.DLL
VWDLGS.DLL
```

b) You can rename RUN.EXE to your liking.   This is recommended.   Try to create a unique name so that it will be unlikely for any File Manager associations to conflict.

c) You will want to edit the startup bitmap VWSIGNON.BMP to your liking using Paintbrush.   The bitmap should not be more than 16 colors, and it shouldn't be much larger than the supplied example. These size restrictions are to minimize the possibility of protection faults happening during program startup.

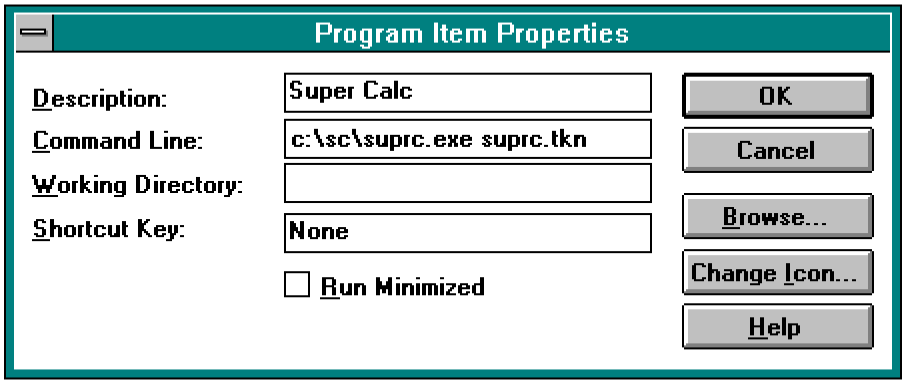
Your vwsignon.bmp file must make mention of:

Portions copyright 1992-1996 Shoptalk Systems
Portions copyright 1991 Digitalk, Inc.

Take a look at the included VWSIGNON.BMP.   The Digitalk, Inc copyright notice is very small.   Feel free to do the same with the above required notices.

d) You may also want to replace the icon for RUN.EXE with one of your own design (icon editor not supplied with Liberty BASIC).

So if you decided to call your program SuperCalc, you might rename RUN.EXE as SUPRC.EXE, and your program's properties might look like:


Program Item Properties

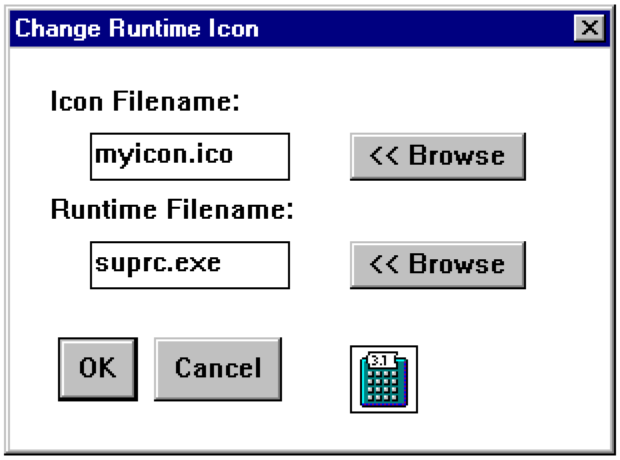
Description: Super Calc

Command Line: c:\sc\suprc.exe suprc.tkn

Working Directory:

Shortcut Key: None

Then use Paintbrush to edit the bitmap vwsignon.bmp to suit your program.
If you designed an icon for your program (a file in *.ico format), you can change the icon for SUPRC.EXE.   Pull down the Setup menu and select Change Runtime Icon to see the dialog box below:


Change Runtime Icon

Icon Filename: myicon.ico

Runtime Filename: suprc.exe

Creating a STARTUP.TKN Program

RUN.EXE will automatically look for a *.TKN file named STARTUP.TKN if you don't specify a *.TKN file to run in the Program Manager item's properties or in your Windows 95 shortcut.   Load File Manager (or Explorer for Windows 95) and double-click on RUN.EXE to see it run the included STARTUP.TKN program.   Here is part of   the source code STARTUP.BAS program (used to make STARTUP.TKN).

```
'STARTUP.BAS
nomainwin
```

```
notice "This is the default startup handler! Take a look at " + _
    "STARTUP.BAS for instructions on using the runtime engine."
end

'ABOUT STARTUP.BAS
'===================================================================
'The Liberty BASIC runtime engine will look for a file called
'STARTUP.TKN if no other *.TKN file is specified in the command
'line.  The STARTUP.TKN file included in the Liberty BASIC
'package is created from this STARTUP.BAS file.  If desired,
'you could create a simple STARTUP.BAS file that would start up
'your application for you.  It might look like this:

nomainwin
run "myprog.tkn"
end
```

Important notice about passing pointers to strings: When passing a pointer to a string that does not have a terminating null character (ASCII 0), Liberty BASIC automatically makes a copy of a string and add the terminating null character. This is because Windows expects strings to be null terminated. The effect of this is that the pointer to your original string is not passed, but instead what is passed is a pointer to a null terminated copy. If Windows then modifies this copied string, you will not be able to read the changes (the GetProfileString API is a an example of a Windows API that modifies a passed string). To get around this, add an ASCII 0 to the string yourself before passing it in this fashion:

```
    stringToPass$ = stringToPass$ + chr$(0)
```

By adding the terminating null character yourself, Liberty BASIC will know not to make a copy, and will instead pass a pointer to the original string.

Liberty BASIC Migration Issues

Liberty BASIC is designed to be a lot like Microsoft BASIC, but there are differences. Some of these differences are incompatibilities, and some are enhancements. By Microsoft BASIC we are referring to GW-BASIC, QBASIC, and QuickBASIC, but not necessarily Visual BASIC.

Here you will find information about:

PRINT Differences

PRINT is -almost- the same.   The comma cannot be used as a formatter.

For example:

```
print a, b, c
```

is valid in Microsoft BASIC, but not in Liberty BASIC.

Also, PRINT USING isn't supported, but there is a using( ) function instead.

```
Microsoft BASIC:    print using "The total is #####.##", ttl
Liberty BASIC:      print "The total is "; using("####.##", ttl)
```

Microsoft BASIC inserts spaces before and after numbers (called padding) when it prints them.   Liberty BASIC doesn't pad, but if you want to, you can add padding spaces in your code.

INPUT Differences

Microsoft BASIC permits this:

```
    input "Give me the next number"; n(x)
```

In Liberty BASIC, you must do this:

```
    input "Give me the next number"; n
    n(x) = n
```

Microsoft BASIC automatically adds a question mark and an extra space onto the end of the prompt (using above as example: Give me the next number? _).   Liberty BASIC doesn't do this.   For example:

```
    input "Give me the next number >"; n
```

appears as:

```
    Give me the next number >_
```

using Liberty BASIC.   If desired, changing the > to a ? will produce the Microsoft BASIC-like effect.

Variable Names

Variable names in Liberty BASIC start with a letter and can contain numeric digits. All characters are significant. Some versions of BASIC only look at the first 8 characters, which would make xylophonePrice and xylophoneWidth the same variable. In addition, upper and lowercase letters are significant, which makes the variable names window and Window different variables.

Here are a few examples of valid variable names:

```
totalAmount
serialNumber
limit3
```

String variables should end with a dollar sign character ($). Liberty BASIC's compiler won't always enforce this rule, but it is best to end string variable names with the $ or sometimes undesireable side effects will appear. Some examples:

```
word1$
firstName$
```

Any variable can contain periods, for example:

```
total.amount
first.name$
serial.number
```

Variable Types

Liberty BASIC only supports two variable types, numeric and string.

The numeric variable holds either an integer or real value, and in this case Liberty BASIC optimizes automatically. If a real loses its non-integer part through some calculation, then Liberty BASIC converts it into an integer to speed up operations. Integers can be enormously large (see factorial.bas). Reals are an eight byte double precision format, but they only display 9 digits of precision by default. More than 9 digits of precision (up to 18 bytes) can be displayed with the using( ) function, like so:

```
  'display the square root of 2 using 18 digits of precision
  print using("#.##################", 2 ^ 0.5)
```

The string variable holds a character string that can be as large as available memory.

NOTE: Variables are actually untyped in Liberty BASIC. A string variable name can contain a numeric value, and a numeric variable name can contain a string. But arrays ARE typed, just as in other BASICs.

Line Numbers

Liberty BASIC lets you use the conventional line numbers if you like. You can also choose not to use them, and use descriptive branch labels instead. For example:

In GWBASIC:

```
  10 print "Let's loop!"
  20 x = x + 1
  30 print x
  40 if x < 10 then 20
  50 print "Done."
  60 end
```

In QBASIC/QuickBASIC

```
      print "Let's loop!"
mainLoop:
      x = x + 1
      print x
      if x < 10 then mainLoop
      print "Done."
      end
```

In Liberty BASIC:

```
      print "Let's loop!"
  [mainLoop]
      x = x + 1
      print x
      if x < 10 then [mainLoop]
      print "Done."
      end
```

You can see here that instead of jumping to line 20, as in the first example, we can instead jump to [mainLoop]. You can name your branch points and subroutines whatever you like. This is similar to QBASIC or QuickBASIC, but not exactly the same. Branch labels in Liberty BASIC need to start and end with [and], respectively, and they can't have any spaces.

Legal branch labels:

```
    [mainLoop]
    [loop1]
    [exceptionHandler]
```

Illegal branch labels:

```
    [main loop]
    mainLoop
    mainLoop:
```

DIM Differences

The DIM statement can only dimension ONE array per statement, so instead of:

```
dim a(20), b(20), c(20)
```

do this:

```
dim a(20) : dim b(20) : dim c(20)
```

File Handles

Microsoft BASIC lets you use only numbers as file handles.   Liberty BASIC lets you use numbers, and it lets you use letters also.

```
   open "autoexec.bat" for input as #1 ' Microsoft BASIC

   open "autoexec.bat" for input as #autoexec    ' Liberty BASIC
```

Now here's the catch.   Whenever you reference the file handle, you MUST use have a # in front of it.

```
   if eof(1) > 0 then 10020 ' Microsoft BASIC

   if eof(#autoexec) > 0 then [reachedEnd]    ' Liberty BASIC
```

Additionally:

```
   print # 1, "buffers = 30"    ' this works fine in Microsoft BASIC
          ^--------this extra space not allowed in Liberty BASIC
```

INPUT$( ) Differences

In Microsoft BASIC, there are two uses for INPUT$().

1) To fetch x number of characters from a sequential file
2) To fetch x number of characters from the keyboard

```
a$ = input$(1, 10) 'Fetch 10 characters from file handle 1
a$ = input$(1) 'Fetch 1 character from the keyboard
```

In Liberty BASIC, there are also two uses for INPUT$().

1) To fetch x number of characters from a sequential file
2) To fetch only 1 character from the keyboard

```
a$ = input$(#1, 10) 'Fetch 10 characters from file handle #1
a$ = input$(1) 'Fetch a single character from the keyboard
```

Using input$(1) to read from the keyboard only works in the main window, and not with any windows created using OPEN.

Boolean Operators

Liberty BASIC supports AND & OR, much like Microsoft BASIC.   These are typically used in IF...THEN statements like so:

```
  if x < limit and userAbort = 0 then [mainLoop]

  if (c$ >= "A" and c$ <= "Z") or (c$ >= "a" and c$ <= "z") then [inRange]
```

Liberty BASIC also supports bitwise operations such as:

```
  print 8 or 16              ' produces 24
  print 7 and 15              ' produces 7
```

Note: The bitwise support was added in LB v1.22

TIME$( ) & DATE$( )

In Microsoft BASIC you get the time or date by using the special variables time$ and date$.   For example:

```
print time$             'this produces the current time
print date$             'this produces the current date
```

In Liberty BASIC it looks like this:

```
print time$()          'this produces the current time
print date$()          'this produces the current date
```

Random Numbers

In Liberty BASIC, random numbers between 0 and 1 are generated with the expression RND(1). Microsoft BASIC will produce the same result with that expression. Apparently Microsoft BASIC also lets you substitute just RND, and it assumes the (1) part in this case. Both of these lines produce the same result in Microsoft BASIC.

```
  x = int(rnd(1)*10)+1   ' this format works with both BASICs
  x = int(rnd*10)+1      ' this format doesn't
```

The second line is not permitted in Liberty BASIC. It will always equal 1, because rnd will be parsed as a numeric variable, and will equal zero, unless it is assigned some other value.

READ, DATA, RESTORE

These are not supported under the current version of Liberty BASIC, but plans to add them are in the works because of popular request.

Troubleshooting

Low memory situtations

Liberty BASIC is a large program, and is a tight fit on 4MB machines (especially using FreeForm). If you find that you are getting low memory errors, try the following:
- Close other running Windows and DOS applications.
- Reduce the size of your Smartdrive disk cache or eliminate it.
- Increase the size of your Windows swapfile.

General Protection Faults

Most general protection faults under Liberty BASIC are caused by:
- Video drivers. A major problem with environments like Windows and OS/2, video drivers are often immature and/or incompletely implemented according to spec. Try to get the most recent version of the Windows drivers for your video card. If it isn't a showstopper for you, try the standard 16 color drivers that come with Windows.
- Low memory (see above). If you are getting a general protection fault in VSTUB.EXE, you need either a bigger swapfile, more physical RAM, or both.

Improperly Redrawn Bitmaps

If you write software in Liberty BASIC that draws bitmaps in graphics windows, and you have trouble with improperly redrawn bitmaps (they draw correctly at first, but if the window is covered, and then uncovered, the bitmaps are not redrawn correctly), the video driver is often the trouble. The author's experience with this problem is that on the same machine, with the same video card, that different results occur just by picking from several different drivers. Using the standard 16 color VGA driver that comes with Windows, things seem to work correctly. Some other video drivers work fine, some cause problems. Always try to get the most up to date drivers for your video card under Windows.

Note: 256 color drivers can give strange results sometimes because Liberty BASIC needs some tweaking in its handling of the Windows pallette.

STRUCT

STRUCT name, field1 as type1 [, field2 as type2, ...]

Description:

This statement builds an single instance of a specified structure that might be required when making some kinds of API/DLL calls.   This does not declare a type of structure, but it creates a single structure.

Here is an example STRUCT statement that builds a Windows rect structure, used in many Windows API calls:

```
    'create the structure winRect
    struct winRect, _
        orgX as ushort, _
        orgY as ushort, _
        extentX as ushort, _
        extentY as ushort
```

To assign a field of a structure some value, treat it much as you would treat a variable, like so:

```
    winRect.orgX.struct = 100
```

And you can use any structure's field like any other variable, like so:

```
    print winRect.orgX.struct
```

or:

```
   newOriginX = offsetX + winRect.orgX.struct
```

When passing a structure in a CALLDLL statement, specify type struct, like in this example:

```
  'WINRECT.BAS - show how to get window position and size
  'and demonstrate how to use the struct statement

  struct winRect, _
      orgX as uShort, _
      orgY as uShort, _
      crnrX as uShort, _
      crnrY as uShort

  open "test me" for window as #win

  open "user.dll" for dll as #user

  hndl =  hwnd(#win)

  calldll #user, "GetWindowRect", _
      hndl as word, _
      winRect as struct, _
      result as void
  print "Upper Left of 'test me': "; winRect.orgX.struct; ", ";
winRect.orgY.struct
  print "Lower Right of 'test me': "; winRect.crnrX.struct; ", ";
winRect.crnrY.struct

  close #user

  input r$

  end
```

Types - Using with STRUCT and CALLDLL

The STRUCT statement requires that each field be typed to specify what type of data it will contain. The CALLDLL statement also requires that each parameter passed be typed. The types are common to both STRUCT and CALLDLL.   Simple data types in Windows programming are often just renamed versions of the types found below.   Substitute the types below as needed.

Here they are:

```
    double          (a double float)
    dword, ulong    (4 bytes)
    handle          (2 bytes)
    long            (4 bytes)
    short           (2 bytes)
    ushort, word    (2 bytes, substitute this for boolean)
   *ptr             (4 bytes, long pointer, for passing strings)
   *struct          (4 bytes, long pointer, for passing structs)
    void, none      (a return type only)

   *types ptr and struct are essentially interchangeable
```

Important notice

If you know you will be passing a negative number as a parameter, first convert the number to a unsigned integer value.   This can be done in Liberty BASIC using the OR operator like so:

```
     variableName  =  -1 or 0    'convert -1 to an unsigned value
```

Liberty BASIC Course

A six week course in Liberty BASIC!   This is the same course taught on-line on the Internet (visit **www.zdu.com**)!

Notice: Look in the tutorial subdirectory of your Liberty BASIC installation for the files mentioned throughout this course lessons below!

Week One Homework

Week 1, homework assignment 1
Liberty BASIC programming course

Refining SALESTAX.BAS

Our SALESTAX.BAS example program provides a good base for us to build on.

Here it is:

```
[start]
    print "Type a dollar and cent amount."
    input "(Press 'Enter' alone for help) ?"; amount
    if amount = 0 then [help]
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax + amount
    goto [start]

[help]
    cls
    print "SALESTAX.BAS Help"
    print
    print "This tax program determines how much tax is"
    print "due on an amount entered and also computes"
    print "the total amount.  The tax rate is 5%."
    print
    input "Press [Enter] to continue."; dummyVariable
    cls
    goto [start]
```

This program calculates a 5% sales tax. It leaves any rounding up to the computer user. We want to extend our SALESTAX program to round up properly and display our amount paid, tax, and total amounts in a way suitable for currency.

Rounding the tax up

Let's define how we will round up tax amounts for our example. Where I live, I pay 5 cents for every whole dollar. The remaining non-whole dollar amount is rounded up or down to the nearest 20 cent amount, and I pay 1 cent for each 20 cents of this rounded result. Let's look at it step by step.

Given a purchase price of $5.45

(a) Strip away the cents from dollars and cents, that gives us $5.

(b) Pay 5 cents for each dollar, or 25 cents ($5 * 5 cents).

(c) Round the 45 cent portion of our purchase price to the nearest multiple of 20, that gives us 40 cents.

(d) Now we will pay 1 cent for every 20 cents in that 40 cent amount for a total of 2 cents (40 / 20 = 2).

(e) Add the 25 cents from (b) to the 2 cents from (d), and we have our total tax amount, 27 cents.

We can compute the value for step (b) using the INT() function.

Let's see how this works:

```
    'demonstration of INT() function
    input "Enter a non integer value (ie. 3.14) ?"; valueA
    let valueB = int(valueA)
    print "The integer part of "; valueA; " is "; valueB
```

To get the cent amount for the above calculations, we subtract the integer part from the entered amount, or valueA - valueB.   Let's see how this works:

```
    'getting the non-integer part of a value
    input "Enter a non integer value (ie. 3.14) ?"; valueA
    let valueB = int(valueA)
    print "The non-integer part of "; valueA; " is "; valueA - valueB
```

Now that we know how to get the cents out of our dollars and cents amount, we need to round it to the nearest 20 cent multiple.   Here is a simple way to round any number to a closest chosen multiple (this works for values greater than 0):

(a) Divide your chosen multiple by two.   We will use this as an adjustment value.   If we round 0.14 to the nearest multiple of 0.20, then our adjustment is 0.10 (0.20 / 2 = 0.10).

(b) Add the adjustment value to the number we want to round.   If our cent amount is 0.45 then our adjusted amount is 0.55.

(c) Now divide the adjusted amount by the multiple we want to round to.   This will give us 2.75 (0.55 / 0.20 = 2.75).

(d) Compute the integer portion of 2.75 and we have 2.   Take this value and multiply it by 0.20 (our chosen multiple), and we get 0.40 (2 * 0.20 = 0.40).

So we see that 0.45 rounded to the nearest multiple of 0.20 is 0.40.   Here is an expanded version of the program above that shows how to round a number in the way outline above:

```
    'getting the non-integer part of a value
    input "Enter a non integer value (ie. 3.14) ?"; valueA
    let valueB = int(valueA)
    valueC = valueA - valueB
    print "The non-integer part of "; valueA; " is "; valueC

    'now round valueC to the nearest multiple of 0.20
    adjustment = 0.20 / 2
    adjustedC = valueC + adjustment
    multiples = int(adjustedC / 0.20)
    roundedC = multiples * 0.20
    print valueC; " rounded to the closest 0.20 is "; roundedC
```

Displaying Monetary Values

When you buy something at the store, the cashier will usually give you a receipt for your purchase.

The product price, sales tax, and total amount paid are formatted something like this:

```
Product Price     4.95
Sales Tax         0.25
----------------------
Total             5.20
```

We want our SALESTAX program to produce information in a similar fashion. Notice that the numbers are all aligned so that their decimal points are one under another. This is called justification. Liberty BASIC provides a function for justifying numeric values called USING(). Here is a quick little program that demonstrates the USING() function:

```
    'display 3 values justified
    valueA = 0.9
    valueB = 120
    print using("#####.##", valueA)
    print using("#####.##", valueB)
    print using("#####.##", valueA + valueB)
```

Running this program produces:

```
    0.90
  120.00
  120.90
```

Notice the string literal "#####.##". The USING() function uses this as a template for formatting a value.

The 5 # characters before the period in "#####.##" tell USING() to place extra spaces in front of the number to ensure that there will be 5 places before the decimal point.

The 2 # characters after the decimal point in "#####.##" tell USING() to add zeros after the decimal point if there are none, or to drop all digits after the second digit following the decimal point.

Loss of Precision

In the realm of computers, real numbers are usually represented in what's called single precision floating point format. This format is not an absolutely accurate way to represent the value of real numbers. Sometimes in the process of executing BASIC code to arrive at a result, some loss of precision will become apparent. This will usually manifest itself as a loss of the minutest amount of a value. For example, a result that should be 1.5 becomes 1.499999.

For many applications, this is not a very significant problem. In the realm of money though, it is a very real problem indeed. If in the course of calculating our sales tax the customer owes $0.24 but the computed result is 0.2399999, it slip through unnoticed. The USING() function would remove the 9's after 0.23, and we would never see it.

The solution is to add 0.001 to monetary values whenever the USING() function is employed. This is important when using PRINT to display results, and also saving information to a disk file (this will be covered later in our course). Here is how that BASIC code would look:

```
    'display 3 values justified
    valueA = 0.9
    valueB = 120
    print using("#####.##", valueA + 0.001)
    print using("#####.##", valueB + 0.001)
    print using("#####.##", valueA + valueB + 0.001)
```

Here's the assignment

Using the techniques we've just covered, extend the SALESTAX.BAS program listed at the start of this text and expand it so that...

- It uses the method of tax calculation described above

- It displays a result in sales receipt fashion like so:

```
Product Price     4.95
Sales Tax         0.25
----------------------
Total             5.20
```

- It corrects its calculations for loss of precision

<ins>Week One Homework Solution</ins>

Week Two - Using Arrays

Using Arrays

Computers are more often than not used to keep track of lists of things. These could be lists of customers, lists of stock #'s, or lists of fonts. A list of things doesn't even have to look like a list. For an example of this, take a look at your favorite video games. If you've ever played PacMan or Doom, the monsters are managed in one kind of list or another (but you never see the list). Arrays are what BASIC (and most other programming languages) uses to keep lists.

To see why we need arrays, let's look at what programming for lists looks like without arrays. Suppose we want to keep track of a list of 10 names. Using only the techniques we've covered so far it would look like this:

```
    'NOARRAY.BAS
    'List handling without arrays

[askForName]  'ask for a name
    input "Please give me your name ?"; yourName$
    if yourName$ = "" then print "No name entered." : goto [quit]

    if name1$ = "" then name1$ = yourName$ : goto [nameAdded]
    if name2$ = "" then name2$ = yourName$ : goto [nameAdded]
    if name3$ = "" then name3$ = yourName$ : goto [nameAdded]
    if name4$ = "" then name4$ = yourName$ : goto [nameAdded]
    if name5$ = "" then name5$ = yourName$ : goto [nameAdded]
    if name6$ = "" then name6$ = yourName$ : goto [nameAdded]
    if name7$ = "" then name7$ = yourName$ : goto [nameAdded]
    if name8$ = "" then name8$ = yourName$ : goto [nameAdded]
    if name9$ = "" then name9$ = yourName$ : goto [nameAdded]
    if name10$ = "" then name10$ = yourName$ : goto [nameAdded]

    'There weren't any available slots, inform user
    print "All ten name slots are already used!"
    goto [quit]

[nameAdded]   'Notify the name add was successful

    print yourName$; " has been added to the list."
    goto [askForName]

[quit]

    end
```

In this example when we enter a name, the computer will look at up to ten variables in turn. If it finds a variable that is an empty string, it will set the value of that variable to be equal to yourName$. Then it will GOTO [nameAdded] and display a notice that the name has been added. If this were part of a larger program, we would execute this code whenever we wanted to add a name, and the name would be inserted into the next available slot.

This technique is manageable only for very small lists.   Imagine how many lines of code we would need to track hundreds or thousands of names in this way!   Arrays provide a special way to make a single line of BASIC code refer to any one variable out of a list of thousands.

Here is our name list example rewritten using a BASIC array.   If it seems unclear to you, try running the debugger on it.

```
    'ARRAYS.BAS
    'List handling with arrays

    dim names$(10)  'set up our array to contain 10 items

[askForName]  'ask for a name
    input "Please give me your name ?"; yourName$
    if yourName$ = "" then print "No name entered." : goto [quit]

    index = 0
[insertLoop]
    'check to see if index points to an unused item in the array
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
    index = index + 1 'add 1 to index
    if index < 10 then [insertLoop] 'loop back until we have counted to 10

    'There weren't any available slots, inform user
    print "All ten name slots already used!"
    goto [quit]

[nameAdded]  'Notify the name add was successful

    print yourName$; " has been added to the list."
    goto [askForName]

[quit]

    end
```

You'll immediately notice the dim names$(10) statement at the start of this example.   This tells BASIC to create an array called names$ with space for 10 items.   Because the name of the array ends with a $ character, this is an array of strings.   An array named without a $ character would contain numeric values.   The first item in an array is referred to by the value 0, and so an array of 10 items starts with item 0 and ends with item 9 (most BASICs work like this, but Liberty BASIC will actually number the array from 0 to 10, giving you eleven items).

The real work is done in this part of the code:

```
    index = 0
[insertLoop]
    'check to see if index points to an unused item in the array
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
    index = index + 1 'add 1 to index
    if index < 10 then [insertLoop] 'loop back until we have counted to 10
```

Here we are using a simple loop to control a search for an empty slot in our array names$.   Take a look at this line:

```
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
```

The if names$(index) = "" part of the line uses the value of index to point to a selected string in the array. When index is 0, we are looking at the first item in the array. When the code loops back and index is 1, it will be looking at the second item in the array, and so on.

In the same line of code, the names$(index) = yourName$ part is used to set the item selected by index. The total effect of the line is to set the selected item in the array to to be yourName$ if that selected array item is an empty string. When this happens it also exits the loop with GOTO [nameAdded].

This is more involved than the technique used in NOARRAY.BAS, but it has the advantage of size. Even if we need to manage 1000 items, we only need to change our code like this (most of the code is left out):

```
    dim names$(1000)
    .
    .
[insertLoop]
    .
    .
    if index < 1000 then [insertLoop] 'loop back until we have counted to
1000
```

If we extend NOARRAY.BAS to manage 1000 names, the resulting program would have one line for each name slot we needed to keep. Our program would be more than 1000 lines long!

Index Out of Bounds

Try running the following mini program.

```
    'done wrong
    dim names$(10)
    print names$(15)
```

You will get an error message saying 'index out of bounds'. This means that you tried to print the contents of an array item that doesn't exist. If you dimension the array to make it large enough, it will work.

```
    'done right
    dim names$(20)
    print names$(15)
```

A similar error will occur if you try to set the item of an array using an index that is too large.

```
    'done wrong again
    dim names$(10)
    names$(15) = "John Doe"
```

Playing the numbers

Numeric arrays work very much like their relative the string array (featured above in ARRAYS.BAS). To demonstrate how numeric arrays work, examine this program that asks for some numbers, adds them up, and calculates the average value.

```
    'AVERAGE.BAS
    'Accept some numbers from the user, then total and average them
    dim numbers(20)
    print "AVERAGE.BAS"
    print

    'loop up to 20 times, getting numbers
    print "Enter up to 20 non-zero values."
    print "A zero or blank entry ends the series."

[entryLoop]  'loop around until a zero entry or until index = 20

    'get the user's entry
    print "Entry "; index + 1;
    input entry

    if entry = 0 then [endSeries]  'quit if entry is zero or blank

    index = index + 1       'add one to index
    numbers(index) = entry  'set the specified array item to be entry
    total = total + entry   'add entry to the total

    if index = 20 then [endSeries]  'if 20 values were entered, exit loop

    goto [entryLoop]  'go back and get another entry

[endSeries]  'entries are finished

    'Set entryCount to index
    entryCount = index
    if entryCount = 0 then print "No Entries." : goto [quit]

    print "Entries completed."
    print
    print "Here are the "; entryCount; " entries:"
    print "-----------------------------"

    'This loop displays each entered value in turn.
    'Notice that we re-use the index variable.  It
    'can be confusing to use a new variable for each
    'new loop.
    index = 0
[displayLoop]
    index = index + 1
    print "Entry "; index; " is "; numbers(index)
    if index < entryCount then [displayLoop]

    'Now display the total and average value
    print
    print "The total is "; total
    print "The average is "; total / entryCount

[quit]

    end
```

Try stepping through this program with the debugger.

****
Challenge exercises
****

1) Add code to ARRAYS.BAS after [quit] to display all the names entered.

2) There's no reason we can't have more than one array in a BASIC program. For an example of this, look at the program file FF12.BAS. Altogether, it uses no fewer than 18 arrays. Some of them are string arrays and some are of the numeric type. Modify the AVERAGE.BAS program so it asks for a name and age for up to 20 people. The names should be managed in a string array (remember ARRAYS.BAS?) that you'll add to the program. When the list of entries is displayed, each name is to be displayed with its age. Then the total and average age will be displayed after this. Call the program AGES.BAS.

Goto - Doing something more than once

Assuming that salestax.bas does what we want (see previous section), it still only does it once. Each time you want to use this handy little program you have to run it again. This can get to be tedious and even error prone (say rubber baby buggy bumpers ten times fast). What we need is a way for our program to go to the beginning and do it over. In BASIC (and in some other languages) the command for doing this is called goto (surprise!).

Knowing that we have to goto some place is not enough. We also need to know where to go. When you hop into your car in a foreign country looking for a food market, you at least know what you are looking for. Liberty BASIC can't ask for directions, so you need to be very precise.

The mechanism that Liberty BASIC uses to mark places that we can goto is called a branch label. This is a lot like a mailing address. When you send a letter or package, you mark it with a known mailing address (hopefully). There is a house or building somewhere marked with that address, and that is where your parcel goes to. So in the same way, you mark the place in your BASIC program where you want it to continue running with a branch label (a mailing address of sorts).

There are two ways to define a branch label in Liberty BASIC. You can use any valid integer number as a branch label, or you can use an easier to remember type which uses letters.

Examples of integer branch labels:

```
     10    150   75    900   5400  etc...
```

Examples of alphanumeric (using letters and numbers) branch labels:

```
     [start]  [loopBack]  [getResponse]  [point1]  etc...
```

Examples of unacceptable branch labels:

```
     [loop back] no spaces allowed
     start       must use brackets
     (point1)    only use square brackets
```

Since no spaces are allowed in the alphanumeric branch labels, it works well to capitalize the first letter in each word when multiple words are used in a branch label. For example [gettimedresponse] is valid, but [getTimedResponse] is much more readable.

So let's pick a branch label for our salestax.bas program. Since we are going to do it over again from the start, we could pick from several reasonable branch label names like perhaps [start], [begin], or [go]. We will use [start] for our program.

Let's add the branch label as shown:

```
[start]
    input "Type a dollar and cent amount "; amount
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax + amount
```

Now we need our goto line. Now that we have our branch label, the correct format for goto is goto [start]. And here's what our program looks like when both a branch label and a goto:

```
[start]
```

```
    input "Type a dollar and cent amount "; amount
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax + amount
    goto [start]
```

Now let's try running this program.   It runs over and over and over, right?   This programming format is called an unconditional loop because it always loops back to repeat the same code no matter what. When we are finished with it, we can close it like any other Windows program by double-clicking on the system menu box.

Next Section: IF . . . THEN - Adding Smarts to Our Tax Program

IF . . . THEN - Adding Smarts to Our Tax Program

The program we designed above will only do one thing for us, no frills.   Let's learn how to add some smarts to our program.   One way that this can be done is with the if . . . then statement.

The if . . . then statement is a direct descendant of those do-it-yourself style instruction manual texts. For example:

Problem: Your car's engine won't turn over

1) Check your battery for correct voltage.
2) If voltage is less then 11 volts then goto to step 13
3) Clean and tighten ground connection.
4) If this doesn't solve the trouble, continue to step 5.
5) Remove the starter.
6) Connect starter directly to battery
7) If starter does not spin then goto step 18
8) Check starter relay.

. . .

13) Charge battery

. . .

18) See chapter 4 on rebuilding the starter unit

Notice how in the above example how you are led smartly through the troubleshooting procedure.   The steps containing the words if and then make it possible to intelligently work through a complex procedure.   In the same way, the if . . . then statement in BASIC makes it possible to add a kind of intelligence to your programs.

Let's see how we can apply this.   Suppose we want the computer to give us the option to display instructions about how to use our tax program.   An easy way to add this ability would be to display instructions whenever a zero value is entered as our dollar amount.

Now, whenever input is used to get a number from the user, if the user doesn't type a number but only presses the [Enter] key, then Liberty BASIC substitutes zero for the variable.   We can exploit this feature in our salestax.bas program.   By checking to see if the variable equals zero after the input statement, we can decide whether or not to display instructions.

Here's what our new program looks like:

```
[start]
    print "Type a dollar and cent amount."
    input "(Press 'Enter' alone for help) ?"; amount
    if amount = 0 then goto [help]
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax + amount
    goto [start]

[help]
    print "This tax program determines how much tax is"
    print "due on an amount entered and also computes"
    print "the total amount.  The tax rate is 5%"
    goto [start]
```

Notice the line "if amount = 0 then goto [help]" in the program above. When Liberty BASIC execute this line, it checks to see if the variable is equal to 0. If the variable equals zero, then the goto[help] statement in the line is executed. Liberty BASIC then begins executing the instructions after the [help] branch label. The if...then statement reads exactly like it executes.

Actually, the goto part of the if . . . then statement is optional. Either of these two forms is acceptable:

```
     if amount = 0 then goto [help]
```
- or -
```
     if amount = 0 then [help]
```

Comparing numbers - The = (equality) operator is only one of several that can be used to make decisions in an if . . . then statement. We can use the if . . . then statement and the (=, <>, <, >, <=, >=) operators to determine whether:

a = b	a is equal to b
a <> b	a is unequal to b
a < b	a is less than b
a > b	a is greater than b
a <= b	a is less than or equal to b
a >= b	a is greater than or equal to b

For example, instead of checking to see if amount was equal to 0 in the above program, we could have checked to see whether it was less than 0.01 (or one cent). For example:

```
    if amount < 0.01 then goto [help]
```

When you run the program above you will probably notice that the things displayed sort of run together. There are things that we can do to neaten up the appearance of a BASIC program. We can add extra blank lines between our printed output to break things up. This is done by using an empty print statement, one for each blank line. We can also clear the window at an appropriate time with the cls statement. Both of these techniques are applied to our tax program in the listing below:

```
[start]
    print "Type a dollar and cent amount."
    input "(Press 'Enter' alone for help) ?"; amount
    if amount = 0 then [help]
    let tax = amount * 0.05
    print "Tax is: "; tax; ". Total is: "; tax + amount
    goto [start]

[help]
    cls
    print "SALESTAX.BAS Help"
    print
    print "This tax program determines how much tax is"
    print "due on an amount entered and also computes"
    print "the total amount.  The tax rate is 5%."
    print
    input "Press [Enter] to continue."; dummyVariable
    print
    goto [start]
```

Notice the line 'input "Press [Enter] to continue."; dummyVariable' in the listing. In this example, we are using an input statement to halt the program, so the instructions can be read. When [Enter] is pressed as instructed, dummyVariable receives the value of what is entered. In this case, only [Enter] is pressed, so dummyVariable gets a value of zero for its data. It really doesn't matter what dummyVariable's data is since we don't use the variable in any calculations elsewhere (hence the name dummyVariable).

Next Section: String Variables

String Variables

So far, the only kind of variables we have used are for holding number values.   There are special variables for holding words and other non-numeric character combinations.   These variables are called string variables (they hold strings of characters*).

*Characters are:
  Letters of the alphabet ;
  Digits 0123456789 ;
  Any other special symbols like: , . < > / ? ; : ' " [ ] { } ` ~ ! @ # $ % ^ & * ( ) + - \ | etc . . .

Let's look at a very simple program using strings:

```
input "Please type your name ?"; name$
print "It's nice to meet you, "; name$
```

This two-line program asks you for your name.   Once you've typed it and pressed [Enter], it responds with:

```
It's nice to meet you, your-name-here
```

Notice one special thing about our string variable name.   It ends with a $ (dollar sign).   In BASIC, when you want to store characters in a variable, you end the variable name with a $.   This makes it a string variable.   As you can see from our program example, you can both input and print with string variables, as we did earlier with our non-string or numeric variables.

We've actually been using strings all along, even before this section about string variables.   Whenever you saw a BASIC program line with words in quotes (for example: print "It's nice to meet you, ") you were looking at what is called a string literal.   This is a way to directly express a string in a BASIC program, exactly the way we type numbers directly in, only with characters instead.   A string literal always starts with a quotation mark and always ends with a quotation mark.   No quotation marks are allowed in between the starting and ending quotation marks (point: string literals cannot contain quotation marks).

NOTE   -   A string can have zero characters.   Such a string is often called an empty string.   In BASIC, an empty string can be expressed in a string literal as two quotation marks without any characters between them.   For example (noCharactersHere$ is the name of our string variable):

```
let noCharactersHere$ = ""
```

Next Section: Some things to do with Strings

Functions

Now that we've covered bringing data into your programs with input, displaying data with print, keeping data in string and numeric variables, and controlling program flow with if . . . then, we will bring one more way to light to flesh out your programs. Functions provide a means for manipulating program data in meaningful ways.

Look this short program:

```
    input "Please type your name ?"; name$
    print "Your name is "; len(name$); " characters long."
```

The second line demonstrates the use of the len( ) function. The len( ) function returns the number of characters in a string. The expression inside of the parenthesis must either be a string literal, a string variable, or an expression that evaluates to be a string. This identifies len( ) as a string function. There are other string functions (for example: val( ), trim$( )). The result returns is a number and can be used in any mathematical expression.

There are numeric functions as well. For example:

```
[start]
    let count = count + 1
    print "The sine of  "; count; " is "; sin(count)
    if count < 45 then goto [start]
```

This simple program lists the sines for the values from 1 to 45. The sin( ) function takes the value of count enclosed in parenthesis and returns the sine (a function in trigonometry, a branch of mathematics) for that value. Just like the len( ) function above, cos( ) and other numeric functions can be used as parts of bigger expressions. We will see how this works just a little further along.

Notice also the way the program counts from 1 to 45. On the first pass, count is equal to zero until it gets to the line let count = count + 1 which makes sets the data for variable count to be one more than its value at that point. Then the program prints the sine of count (the sine of one, in other words). After this, the line if count < 45 goto [start] checks to see if the data for count is less than 45. If it is, then BASIC goes back to the branch label [start] to do it again. This happens over and over until count reaches a value of 45, and then it doesn't go back to [start] again, but instead having no more lines of code to run, the program stops.

Going back to execute code over again is called looping. We saw this earlier when we first used the goto statement. In our first use of goto, the program always looped back. In this newest example program we see going back to execute code over again, but based on a condition (in this case whether count is less than 45). This is called conditional looping (you guessed it, the looping that always happens is called unconditional looping, or infinite looping).

Next Section: Documenting BASIC Code

Documenting BASIC Code

When writing very short and simple BASIC programs, it isn't usually difficult to grasp how they work when reading them days or even weeks later. When a program starts to get large then it can be much harder. There are things that the programmer (yes, you) can do to make BASIC programs more understandable.

VARIABLES - Liberty BASIC makes it easy to give your variables very meaningful names. Since a variable name can be as long as you like and because Liberty BASIC lets you use upper and lower case letters, variable names can be very meaningful.

For example:

```
let c = (a^2 + b^2) ^ 0.5
```

could better be expressed:

```
let lengthOfCable = (distanceFromPole ^ 2 + heightOfPole ^ 2) ^ 0.5
```

Both are valid Liberty BASIC code, but the second is easier to read and maintain.

BRANCH LABELS - Make sure that when you use goto that your branch labels describe the kind of activity your BASIC program performs after the label. For example if you are branching to a routine that displays help then use [help] as your branch label. Or if you are branching to the end of your program you might use [endProgram] or [quit] as branch labels.

COMMENTING CODE - BASIC also has a built in documentation feature that lets you add as much commentary as you like in the language of your choice. The rem (short for remark) statement lets you type whatever you like after it (you can even misspell or type gobbledy-gook, it doesn't care!). For example:

```
[askForName]

    rem  Ask for the user's name
    input "What is your name ?" ; yourName$

    rem  If the user didn't type anything, then ask again
    if yourName$ = "" then goto [askForName]
```

Notice how a rem statement was added before the input statement and before the if . . . then statement to describe what they should do. Liberty BASIC just skips over these lines, but a human reader finds this kind of documentation very helpful.

Also see the way that blank lines were added between the different parts of the program? These help to group things together, making the program easier to read.

A more elegant form of the rem statement uses the ' (apostrophe, the key just to the left of the Enter key). Instead of typing rem, substitute the ' like so:

```
[askForName]

    '  Ask for the user's name
    input "What is your name ?" ; yourName$

    '  If the user didn't type anything, then ask again
    if yourName$ = "" then goto [askForName]
```

Most people consider this to be more readable than using rem, and it works the same. One extra thing that you can do only with the apostrophe version of rem is to hang it off the end of whatever line you are commenting. For example:

```
[askForName]

    input "What is your name ?" ; yourName$      '  Ask for the user's name
    if yourName$ = "" then goto [askForName]  '  The user didn't type
anything. Ask again
```

This optional, but it saves screen space and many prefer it.

Learning to document the programs you write takes practice. Try to develop a consistent style. Everyone does it differently and there isn't a right or wrong way to do it. Smaller programs may not need any documentation at all. Programs that you intend to share with others should probably be thoroughly documented.

Next Section: Let's Write a Program - HILO.BAS

Let's Write a Program - HILO.BAS

Now we will write a simple game using all of the concepts described in this tutorial. These include:

input statement
print statement
let statement
variables
goto statement
conditional branching with if . . . then
functions
documenting

THE GAME - HILO.BAS

Hi-Lo is a simple guessing game. The computer will pick a number between 1 and 100. Our job is to guess the number in as few guesses as we can. When we guess, the computer will tell us to guess higher or to guess lower depending on whether we guessed too high or too low. When we finally get it, the computer will tell us how many guesses it took.

Let's outline how our program will work before we begin to write code:

```
(1) Pick a number between 1 and 100
(2) Print program title and give some instructions
(3) Ask for the user to guess number
(4) Tally the guess
(5) If the guess is right go to step (9)
(6) If the guess is too low tell the user to guess higher
(7) If the guess is too high tell the user to guess lower
(8) Go back to step (3)
(9) Beep and tell the user how many guess it took to win
(10) Ask the user whether to play again
(11) If the user answers yes then clear the guess tally and goto step (1)
(12) Give the user instructions on how to close the game window
(13) End the program
```

When we write an outline for a computer program like we did here, the resulting outline is often called pseudocode, which is a fancy name for false code. This can be a useful tool for planning out software before it is written, and it can be very helpful in developing ideas before code is actually written.

Now we are going to take each of the steps above and write BASIC code for each step. We will document the code to explain its purpose:

(1) Pick a number between 1 and 100

```
    ' Here is an interactive HI-LO
    ' Program

[start]
    guessMe = int(rnd(1)*100) + 1
```

The first couple of lines are just ' remark statements to give a brief desciption for the program. The [start] branch label is equivalent to calling this part of the program step

Now we have the code that picks the number. We use two functions here to accomplish this task:

The rnd( ) function is the key to this line of code.   It picks a random (or nearly random) number greater than 0 and less than 1 (for example 0.3256).   Then we multiply this by 100 with the * operator to get a number between greater than 0 but less then 100 (0.3256 times 100 would be 32.56) ;

The int( ) function removes the fractional part to leave only the integer part of the number (the 0.56 part of 32.56 would be removed to leave only 32).

Then we add 1 to this.   This is necessary because we want to pick a number as small as 1 and as large as 100.   The rnd( ) function only gives us a number as large as 0.9999999 and not as large as 1. If you multiply 0.9999999 by 100, the biggest number you can get is 99.99999, and this is not big enough so we add one.

When we have picked the number, we assign its value to the variable guessMe.

2) Print program title and give some instructions

```
    ' Clear the screen and print the title and instructions
    cls
    print "HI-LO"

    print

    print "I have decided on a number between one"
    print "and a hundred, and I want you to guess"
    print "what it is.  I will tell you to guess"
    print "higher or lower, and we'll count up"
    print "the number of guesses you use."

    print
```

This very simple part of the program wipes the window clean and prints the title HI-LO and some instructions.   Notice the use of blank print statements to add space between the title and the instructions and after the instructions also.

3) Ask for the user to guess number

```
[ask]
    ' Ask the user to guess the number and tally the guess
    input "OK.  What is your guess"; guess
```

Here the branch label [ask] will let us go back here later if the user needs to be asked to guess again. Then we use the input statement to ask the user for a guess.   The user's guess is then placed in the guess (what else?) variable.

4) Tally the guess

```
    ' Now add one to the count variable to count the guesses
    let count = count + 1
```

Now we take the value of count and add one to it.   Each time this code is performed, count's value will increase by one.

5) If the guess is right go to step (9)

```
    ' check to see if the guess is right
    if guess = guessMe then goto [win]
```

This line compares the variable guess with the variable guessMe. If they are equal, then we goto [win], which is equivalent to . . . go to step (9) in our outline.

6) If the guess is too low tell the user to guess higher

```
    ' check to see if the guess is too low
    if guess < guessMe then print "Guess higher."
```

This line compares the variable guess with the variable guessMe. If guess is less than guessMe, then display the text "Guess higher."

7) If the guess is too high tell the user to guess lower

```
    ' check to see if the guess is too high
    if guess > guessMe then print "Guess lower."
```

This line compares the variable guess with the variable guessMe. If guess is greater than guessMe, then display the text "Guess lower."

8) Go back to step 3

```
    ' go back and ask again
    goto [ask]
```

9) Beep and tell the user how many guess it took to win

```
[win]
    ' beep once and tell how many guesses it took to win
    beep
    print "You win!  It took"; count; "guesses."

    ' reset the count variable to zero for the next game
    let count = 0
```

This is the code our game executes when the player wins. The beep statement rings the terminal bell once. Then the print statement says that the game is won and how many guesses it took. Finally, the let statement resets the count variable to zero for the next game.

10) Ask the user whether to play again

```
    ' ask to play again
    input "Play again (Y/N)"; play$
```

This input statement asks whether or not to play again. The resulting string is stored in the string variable called play$.

11) If the user answers yes then goto step 1

```
    if instr("YESyes", play$) > 0 then goto [start]
```

This if . . . then statement uses the instr( ) function to determine whether the player answered "Y", "y", "YES", or "yes". The instr( ) function checks to see if the string in play$ is found anywhere in the string literal "YESyes". If it is, then instr( ) returns the position, which is then compared with 0 using the > operator. If the contents of play$ are found in "YESyes", then the value returned from instr( ) will be greater than zero, so that goto [start] will be executed, and the game will be restarted.

12) Give the user instructions on how to close the game window

```
    print "Press ALT-F4 to close this window."
```

Since the player did not wish to play again, we will display instructions on how to close the game window.

13) End the program

```
    end
```

It is good practice place the end statement as the end of your BASIC programs. It can also be used at any place where you want the program to stop running.

Next Section: Complete listing for HILO.BAS

Complete listing for HILO.BAS

Here is the complete listing for HILO.BAS so that you can just copy and paste it into Liberty BASIC to run it.

```
    ' Here is an interactive HI-LO
    ' Program

[start]
    guessMe = int(rnd(1)*100) + 1

    ' Clear the screen and print the title and instructions
    cls
    print "HI-LO"

    print

    print "I have decided on a number between one"
    print "and a hundred, and I want you to guess"
    print "what it is.  I will tell you to guess"
    print "higher or lower, and we'll count up"
    print "the number of guesses you use."

    print

[ask]
    ' Ask the user to guess the number and tally the guess
    input "OK.  What is your guess"; guess

    ' Now add one to the count variable to count the guesses
    let count = count + 1

    ' check to see if the guess is right
    if guess = guessMe then goto [win]
    ' check to see if the guess is too low
    if guess < guessMe then print "Guess higher."
    ' check to see if the guess is too high
    if guess > guessMe then print "Guess lower."

    ' go back and ask again
    goto [ask]

[win]
    ' beep once and tell how many guesses it took to win
    beep
    print "You win!  It took"; count; "guesses."

    ' reset the count variable to zero for the next game
    let count = 0

    ' ask to play again
    input "Play again (Y/N)"; play$
    if instr("YESyes", play$) > 0 then goto [start]

    print "Press ALT-F4 to close this window."
```

```
end
```

Some things to do with Strings

Just as you can manipulate numbers in a computer programming language by adding, subtracting, multiplying, and dividing, (and more!), we can also manipulate strings

Adding strings - We can add (or concatenate) two or more strings together in BASIC like so:

```
    input "What is your first name ?"; firstName$
    input "What is your last name ?"; lastName$
    let fullName$ = firstName$ + " " + lastName$
    print "Your full name is: "; fullName$
```

In this short program, we input two strings, your first and last name. Then we concatenate the string in firstName$ with the string literal " " (a single space between two quotes) and with lastName$. The result is made the data for the string variable fullName$, which we then print out.

Comparing strings - We can compare strings with each other just as we can compare numbers. This means that we can use the if . . . then statement and the (=, <>, <, >, <=, >=) operators to determine whether:

```
    a$ = b$      a$ is equal to b$
    a$ <> b$     a$ is unequal to b$
    a$ < b$      a$ is less than b$
    a$ > b$      a$ is greater than b$
    a$ <= b$     a$ is less than or equal to b$
    a$ >= b$     a$ is greater than or equal to b$
```

When comparing strings, a string is considered to be equal to another string when all the characters in one string are exactly the same in both strings. This means that even if they both print the same onto the screen, they can still be unequal if one has an invisible space on the end, and the other doesn't. For example:

```
    a$ = "Hello"
    b$ = "Hello "
    print "a$ is "; a$
    print "b$ is "; b$
    if a$ = b$ then goto [areTheSame]
    print "a$ and b$ are not the same"
    goto [end]
[areTheSame]
    print "a$ and b$ are the same"
[end]
```

When the line if a$ = b$ then goto [same] is performed, the result is not to goto [same], because even though if a$ and b$ were printed they would look the same, they are not actually the same.

Next Section: Functions

Week One Homework Solution

```
    'Week 1, homework solution 1
    'Liberty BASIC programming course
    'Copyright 1996 Shoptalk Systems
    'All Rights Reserved

[start]

    'Ask for a dollar and cent amount
    print "Type a dollar and cent amount."
    input "(Press 'Enter' alone for help) ?"; amount

    'Display help if user pressed [Enter]
    if amount = 0 then [help]

    'Calculate tax for the nearest 20 cent multiple
    dollars = int(amount)
    dollarsTax = dollars * 0.05
    cents = amount - dollars
    adjustment = 0.10
    centsAdjusted = cents + adjustment
    centsRounded = int(centsAdjusted / 0.20) * 0.20
    centsTax = (centsRounded / 0.20) / 100
    tax = dollarsTax + centsTax

    'Display the amount, tax, and total
    'Add 0.001 to compensate for any loss of precision because
    'we are dealing with monetary values.
    print "Product Price $"; using("#####.##", amount + 0.001)
    print "Sales Tax      "; using("#####.##", tax + 0.001)
    print "------------------------"
    print "Total          "; using("#####.##", tax + amount + 0.001)
    print
    goto [start]

[help]
    cls
    print "SALESTAX.BAS Help"
    print
    print "This tax program determines how much tax is"
    print "due on an amount entered and also computes"
    print "the total amount.  The tax rate is 5%."
    print
    input "Press [Enter] to continue."; dummyVariable
    cls
    goto [start]
```

Week Two Homework

Week 2 Homework
Liberty BASIC programming course

FOR/NEXT Loops

Up to this point most of the loops that we've used in our code have had this form:

```
    variable = 0
[loopPoint]
    'put some code in here
    variable = variable + 1
    if variable < 10 then [loopPoint]
```

This will loop 10 times. The first time through the loop the value of variable will be 0, and the tenth time through the loop it will be 9. This kind of coding does work, but it can look complicated and it takes some care to make sure it is coded properly (it's easy to make a mistake). Now let's take a look at an alternative:

```
    for variable = 0 to 9
        'put some code here
    next variable
```

This is called a FOR/NEXT loop. The code that is inserted between the FOR and NEXT statement will be executed 10 times. The value of variable will be 0 the first time through and 9 the last time.

We can use this kind of code to do something a specifed number of times or to do something using a range of values (for example 0 to 9).

FOR/NEXT loops require fewer lines of code. When programming, it is usually best to use the simplest possible way to code any solution. This kind of simplicity doesn't come without a price. There are things you can do with the first kind of looping that you cannot do with a FOR/NEXT loop.

Look at this example:

```
    variable = 0
[loopPoint]
    'put your code here
    if someCondition = 1 then [specialException]
    variable = variable + 1
    if variable < 10 then [loopPoint]
```

Notice the line of code that branches to [specialException] if someCondition is equal to 1. This is perfectly acceptable coding practice. Now look at this example:

```
    for variable = 0 to 9
        'put your code here
        if someCondition = 1 then [specialException]
    next variable
```

This is not acceptable. Liberty BASIC expects the loop to finish properly. It is OK to use GOSUB to call a subroutine from inside a FOR/NEXT loop because when the subroutine returns, execution will resume inside the loop and it will be allowed to finish properly.

A good question to ask when deciding whether to use a FOR/NEXT loop is "Do I know how many times this loop will be executed?" In our example above we don't know because at any time someCondition could suddenly be equal to
1, and then our loop is finished.

You can cheat. If you set the value of variable to its finishing value (or limit), the loop will think it's all done and it will quit. For example the following code will only count to 5.

```
    for x = 0 to 9
        print x
        if x = 5 then x = 9
    next x
```

It's also worth noting that FOR/NEXT loops don't need to use integer values. Here is an example that counts from 0 to 1 in steps of less than 1.

```
    for i = 0 to 1 step 0.04
        print i
    next i
```

By using the word STEP and a value we specified what value to add to our variable (i in this case). We can count backwards using a similar technique.

```
    for j = 10 to 1 step -1
        print j
    next j
```

Finally, if a starting value for the variable in our FOR/NEXT loop is greater than its limit, the code between the FOR and NEXT statements will be skipped over. The following code will not display any numbers at all.

```
    for x = 1 to 0
        print x  'this line will not be executed
    next x
```

Similarly...

```
    for x = 0 to 1 step -1
        print x  'this line will not be executed
    next x
```

Here is our ARRAYS.BAS program taken from our Week 2 Solutions to Challenge Exercises. We've modified it by replacing one of the loops with a FOR/NEXT loop. Study it carefully.

```
    'ARRAYS.BAS
    'List handling with arrays
    'Here is a version that displays the names entered
    'after [quit]

    dim names$(10)   'set up our array to contain 10 items

[askForName]  'ask for a name
    input "Please give me your name ?"; yourName$
    if yourName$ = "" then print "No name entered." : goto [quit]

    index = 0
[insertLoop]
    'check to see if index points to an unused item in the array
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
    index = index + 1 'add 1 to index
    if index < 10 then [insertLoop] 'loop back until we have counted to 10

    'There weren't any available slots, inform user
    print "All ten name slots already used!"
    goto [quit]

[nameAdded]  'Notify the name add was successful

    print yourName$; " has been added to the list."
    highestSlot = index  'this is a new line of code
    goto [askForName]

[quit]

    'display all the entered names
    print
    print "Here is a list of the names entered:"
    print "-----------------------------------"
    for index = 0 to highestSlot
         if names$(index) <> "" then print names$(index)
    next index

    end
```

Here's the assignment

1) In similar fashion to our modification of ARRAYS.BAS (above), modify the AGES.BAS program below, replacing appropriate looping code with FOR/NEXT loops.

```
    'AGES.BAS
    'Accept some names and ages from the user, then total and average them
    dim numbers(20)
    dim names$(20)
    print "AGES.BAS"
    print

    'loop up to 20 times, getting numbers
    print "Enter up to 20 non-zero values."
    print "A zero or blank entry ends the series."

[entryLoop]  'loop around until a zero entry or until index = 20

    'get the user's name and age
    print "Entry "; index + 1;
    input name$
    if name$ = "" then [endSeries]  'quit if name$ is blank
    print "Age   ";
    input age
    index = index + 1       'add one to index
    names$(index) = name$   'set the specified array item to be name$
    numbers(index) = age    'set the specified array item to be age
    total = total + age     'add entry to the total

    if index = 20 then [endSeries]  'if 20 values were entered, exit loop

    goto [entryLoop]  'go back and get another entry

[endSeries]  'entries are finished

    'Set entryCount to index
    entryCount = index
    if entryCount = 0 then print "No Entries." : goto [quit]

    print "Entries completed."
    print
    print "Here are the "; entryCount; " entries:"
    print "-----------------------------"

    'This loop displays each entered value in turn.
    'Notice that we re-use the index variable.  It
    'can be confusing to use a new variable for each
    'new loop.
    index = 0
[displayLoop]
    index = index + 1
    print "Entry "; index; " is "; names$(index); ", age "; numbers(index)
    if index < entryCount then [displayLoop]

    'Now display the total and average value
    print
    print "The total age is "; total
    print "The average age is "; total / entryCount

[quit]

    end
```

2) Add a routine onto the end of AGES.BAS that asks the user for a name.   Then using a FOR/NEXT loop searches for that name in the names$ array and if the name is found displays the age for that name.

Homework Solution

Week 2 course material: Solutions to challenge exercises
Liberty BASIC programming course

1) Add code to ARRAYS.BAS after [quit] to display all the names entered.

```
    'ARRAYS.BAS
    'List handling with arrays
    'Here is a version that displays the names entered
    'after [quit]

    dim names$(10)  'set up our array to contain 10 items

[askForName]  'ask for a name
    input "Please give me your name ?"; yourName$
    if yourName$ = "" then print "No name entered." : goto [quit]

    index = 0
[insertLoop]
    'check to see if index points to an unused item in the array
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
    index = index + 1 'add 1 to index
    if index < 10 then [insertLoop] 'loop back until we have counted to 10

    'There weren't any available slots, inform user
    print "All ten name slots already used!"
    goto [quit]

[nameAdded]  'Notify the name add was successful

    print yourName$; " has been added to the list."
    goto [askForName]

[quit]

    'display all the entered names
    print
    print "Here is a list of the names entered:"
    print "-------------------------------------"
    index = 0

[displayLoop]

    if names$(index) <> "" then print names$(index)
    index = index + 1
    if index < 10 then [displayLoop]

    end
```

2) Modify the AVERAGE.BAS program so it asks for a name and age for up to 20 people.   The names

should be managed in a string array (remember ARRAYS.BAS?) that you'll add to the program. When the list of entries is displayed, each name is to be displayed with its age. Then the total and average age will be displayed after this. Call the program AGES.BAS.

```
    'AGES.BAS
    'Accept some names and ages from the user, then total and average them
    dim numbers(20)
    dim names$(20)
    print "AGES.BAS"
    print

    'loop up to 20 times, getting names and ages
    print "Enter up to 20 names with ages."
    print "A blank name entry ends the series."

[entryLoop]  'loop around until a zero entry or until index = 20

    'get a name and age
    print "Name "; index + 1;
    input name$
    if name$ = "" then [endSeries]  'quit if name$ is blank
    print "Age   ";
    input age

    index = index + 1       'add one to index
    names$(index) = name$   'set the specified array item to be name$
    numbers(index) = age    'set the specified array item to be age
    total = total + age     'add entry to the total

    if index = 20 then [endSeries]  'if 20 values were entered, exit loop

    goto [entryLoop]  'go back and get another entry

[endSeries]  'entries are finished

    'Set entryCount to index
    entryCount = index
    if entryCount = 0 then print "No Entries." : goto [quit]

    print "Entries completed."
    print
    print "Here are the "; entryCount; " entries:"
    print "-----------------------------"

    'This loop displays each entered value in turn.
    'Notice that we re-use the index variable.  It
    'can be confusing to use a new variable for each
    'new loop.
    index = 0
[displayLoop]
    index = index + 1
    print "Entry "; index; " is "; names$(index); ", age "; numbers(index)
    if index < entryCount then [displayLoop]

    'Now display the total and average value
    print
    print "The total age is "; total
    print "The average age is "; total / entryCount

[quit]

    end
```

Week Three - Reading and Writing Sequential Files

Week 3 course material
Liberty BASIC programming course

Reading and Writing Sequential Files

Now we will learn about how to work with disk files. With few exceptions, all personal computers have at least one floppy disk drive and one hard disk drive. Liberty BASIC provides ways to write information to disk files on these devices, and we can take advantage of this when we write our programs.

There are two ways to read and write files in Liberty BASIC. One is called sequential and the other is called random access. We will use the sequential method for our examples here. The reason it is called sequential is that
when reading or writing we start at the beginning of the file and work one item at a time to the end. An item can be words or characters separated by commas, or an item can be a complete line of data.

Let's familiarize ourselves with a few Liberty BASIC statements that help us work with files.

OPEN

The OPEN statement causes Liberty BASIC to open a file. A file must be opened if we want to write into it or read from it. There are several ways to open any file for sequential access. These 'ways' are called modes.

The OUTPUT mode: The OUTPUT mode is for writing to a file. This is what an OPEN statement for OUTPUT looks like:

```
    open "myfile.txt" for output as #myHandle
```

You can see the OUTPUT mode is specified. The last item on the line is #myHandle. It is a name (called a file handle) given to Liberty BASIC to use for the open file. Any code that writes to this file must include a reference to #myHandle. This is so that Liberty BASIC knows which file to write to.

A file handle starts with a # character followed by any word or sequence of characters (using letters and digits). It is best to choose handles that make it easy for you to remember which file you are working with while writing your program. Some examples of valid file handles are:

```
    #1
    #abc
    #dataFile
    #customers
```

You cannot have more than one file open at a time that uses the same file handle or your program will terminate with an error.

The INPUT mode: There is also a mode for reading sequentially from a file. This mode is called INPUT. Here is an example of an OPEN statement for INPUT:

```
    open "myfile.txt" for input as #myHandle
```

CLOSE

The CLOSE statement is used for closing open files when we are done reading or writing them. This is a required operation when working with files and it is a very important thing to remember when writing programs.

Here is how OPEN and CLOSE work together:

```
    open "myfile.txt" for output as #myHandle
    'put some code in here that writes to #myHandle
    close #myHandle
```

Another thing to remember is that a file opened for one mode must first be closed before it can be opened for a different mode. If you are going to read from a file you've just written to, you must close the file and reopen
it, like so:

```
    open "myfile.txt" for output as #myHandle
    'put some code in here that writes to #myHandle
    close #myHandle

    open "myfile.txt" for input as #myHandle
    'put some code in here that reads from #myHandle
    close #myHandle
```

PRINT

We've already seen how the PRINT statement can display text into a window on the screen. PRINT can also be used to write into a file opened for sequential OUTPUT. Here is an example:

```
    open "myfile.txt" for output as #myHandle
    print #myHandle, "Hello"
    print #myHandle, "World!"
    close #myHandle
```

This little program produces a file containing two lines of text (each could be considered an item, see above). Type the code in and run it. When the program finishes executing, open Windows Notepad on MYFILE.TXT to see the
result!

Let's Take It For A Spin

Now we'll modify the AGES.BAS program from out WK2SOL.TXT file so that it saves the names and ages that we enter into a file. Take a look at this modified program:

```
    'AGES.BAS
    'Accept some names and ages from the user, then total and average them
    dim numbers(20)
    dim names$(20)
    print "AGES.BAS"
    print

    'loop up to 20 times, getting numbers
    print "Enter up to 20 non-zero values."
    print "A zero or blank entry ends the series."

[entryLoop]  'loop around until a zero entry or until index = 20

    'get the user's name and age
    print "Entry "; index + 1;
    input name$
    if name$ = "" then [endSeries]  'quit if name$ is blank
    print "Age   ";
    input age

    index = index + 1       'add one to index
    names$(index) = name$   'set the specified array item to be name$
    numbers(index) = age    'set the specified array item to be age
    total = total + age     'add entry to the total

    if index = 20 then [endSeries]  'if 20 values were entered, exit loop

    goto [entryLoop]  'go back and get another entry

[endSeries]  'entries are finished

    'Set entryCount to index
    entryCount = index
    if entryCount = 0 then print "No Entries." : goto [quit]

    print "Entries completed."
    print
    print "Here are the "; entryCount; " entries:"
    print "-----------------------------"

    'This loop displays each entered value in turn.
    'Notice that we re-use the index variable.  It
    'can be confusing to use a new variable for each
    'new loop.
    for index = 1 to entryCount
      print "Entry "; index; " is "; names$(index); ", age "; numbers(index)
    next index

    '*** New code starts here ***

    'Write the data into ages.dat
    open "ages.dat" for output as #ages
    for index = 1 to entryCount
      print #ages, names$(index)
      print #ages, numbers(index)
    next index
    close #ages

    '*** New code ends here ***

    'Now display the total and average value
    print
    print "The total age is "; total
    print "The average age is "; total / entryCount

[quit]
```

```
    end
```

Type the new program lines in. Run the program and enter a few names and ages. When the program finishes executing, open the file with Notepad and you'll the data you entered. It should look something like:

```
Tom Jones
52
Victor Krueger
39
Sue White
64
```

Let's see how our newly added code works.

1) First we open the file AGES.DAT with the OPEN statement. It is opened for OUTPUT (writing) and its file handle is #ages.

```
    open "ages.dat" for output as #ages
```

2) This sets up a FOR/NEXT loop.

```
    for index = 1 to entryCount
```

3) Now we print a name and age, each on a separate line.

```
    print #ages, names$(index)
    print #ages, numbers(index)
```

4) Here's the back end of our FOR/NEXT loop. Loop back until index equals entryCount.

5) Now we will close AGES.DAT.

```
    close #ages
```

Reading from a file

Now that we've written information to a disk file, we are going to read that information back into our program. This is done using the INPUT statement. Just as we saw PRINT used to display information in a window and to write
information to a disk file, INPUT can be used to get keyboard input, or to get information from a disk file.

To read from a file, it must be opened using the INPUT mode. The OPEN statement is used like so:

```
    open "ages.dat" for input as #ages
```

Our INPUT statement for reading from a file looks a lot like the PRINT statement above:

```
    input #ages, var$
```

Notice we use the file handle, and then we specify a variable name to read into. In a program that reads a list of items, an INPUT statement like the one above would be placed inside of a loop. In this way each item in the
file can be read in turn and stored in an array.

An important point is that that we don't always know how many items have been written to AGES.DAT. This means we don't know when to stop looping around and reading items from the file. One solution is to add a PRINT
statement to the program that creates the file. This PRINT statement would write the number of items at the start of the file, like so:

```
    'Write the data into ages.dat
    open "ages.dat" for output as #ages
    print #ages, entryCount
    for index = 1 to entryCount
      print #ages, names$(index)
      print #ages, numbers(index)
    next index
    close #ages
```

Then all we would need to do is read that number first, and then loop that many times to read each name and age. I will tackle it from a different direction though, because I want to introduce the EOF() function.

The EOF() function stands for End Of File. For a given file handle, it will return 0 if we are not at the end of file, and -1 if we are at the end of file. For example:

```
    open "ages.dat" for input as #ages
    if eof(#ages) = 0 then print "NOT AT END OF FILE"
    close #ages
```

The above code would print NOT AT END OF FILE because we haven't read all the way to the end of the file.

Here is a version of AGES.BAS that reads it's information from the file we created above and uses EOF() to check for the end of file.

```
    'AGES_IN.BAS
    'Read names and ages from AGES.DAT, then total and average them.
    'This version doesn't write the data back out to the file.
    dim numbers(20)
    dim names$(20)
    print "AGES_IN.BAS"
    print

    print "Reading AGES.DAT..."

    'open ages.dat
    open "ages.dat" for input as #ages
[entryLoop]  'loop around until end of file or until index = 20

    'test for the end of file
    if eof(#ages) = -1 then [endSeries]

    'get the user's name and age
    input #ages, name$
    input #ages, age

    index = index + 1       'add one to index
    names$(index) = name$   'set the specified array item to be name$
    numbers(index) = age    'set the specified array item to be age
    total = total + age     'add entry to the total

    if index = 20 then [endSeries]  'if 20 values were entered, exit loop

    goto [entryLoop]  'go back and get another entry

[endSeries]  'entries are finished

    'close ages.dat
    close #ages

    'Set entryCount to index
    entryCount = index
    if entryCount = 0 then print "No Entries." : goto [quit]

    print "Entries completed."
    print
    print "Here are the "; entryCount; " entries:"
    print "-----------------------------"

    'This loop displays each entered value in turn.
    'Notice that we re-use the index variable.  It
    'can be confusing to use a new variable for each
    'new loop.
    for index = 1 to entryCount
      print "Entry "; index; " is "; names$(index); ", age "; numbers(index)
    next index

    'Now display the total and average value
    print
    print "The total age is "; total
    print "The average age is "; total / entryCount

[quit]

    end
```

Challenge Exercise

Create a modified ARRAYS.BAS using the code below so that it keeps its list in a disk file named ARRAYS.DAT.   The program must read its list from ARRAYS.DAT before asking for additional names. When the user is done
entering names, or if all the slots are filled, the list will be written to ARRAYS.DAT before the program ends.   See the source code below for some clues.

```
    'ARRAYS2.BAS
    'List handling with arrays and file input/output.
    'This version stores more than 10 names.

    dim names$(50)  'set up our array to contain 50 items

    'Insert code that reads ARRAYS.DAT into the array names$.
```

```
[askForName]  'ask for a name
    input "Please give me your name ?"; yourName$
    if yourName$ = "" then print "No name entered." : goto [quit]

    index = 0
[insertLoop]
    'check to see if index points to an unused item in the array
    if names$(index) = "" then names$(index) = yourName$ : goto [nameAdded]
    index = index + 1 'add 1 to index
    if index < 50 then [insertLoop] 'loop back until we have counted to 50

    'There weren't any available slots, inform user
    print "All ten name slots already used!"
    goto [quit]

[nameAdded]  'Notify the name add was successful

    print yourName$; " has been added to the list."
    goto [askForName]

[quit]

    'display all the entered names
    print
    print "Here is a list of the names entered:"
    print "-------------------------------------"

    for index = 0 to 49
        if names$(index) <> "" then print names$(index)
    next index

    'Insert code that saves the names$ array to ARRAYS.DAT.

    end
```

Week Three Homework

Week 3 Homework 1
Liberty BASIC programming course

In our first Week 3 installment (WK3CM1.TXT) we covered the OUTPUT and INPUT file modes. These are for writing and reading files, respectively. This part of the lesson covers a new file mode called APPEND. We will also learn how to use the LINE INPUT statement, and we will also use a few other new Liberty BASIC statements in passing.

The APPEND file mode

There is a limitation in the OUTPUT file mode. The OUTPUT mode is only good for creating new files because it always deletes the contents of any file opened when it is used. For example:

```
    'open a file and print some information into it
    open "numbers.txt" for output as #out
    for x = 1 to 10
        print #out, x
    next x
    close #out

    'now rewrite the file, replacing its original contents
    open "numbers.txt" for output as #out
    for x = 11 to 20
        print #out, x
    next x

    end
```

Run the above program and open Notepad on NUMBERS.TXT. You'll see that only the numbers 11 to 20 will be in the file. The only way to retain the original information in the file and add new information using the OUTPUT mode would be to read the original information and insert it into a new file. Then the new information is inserted into the still open file. Here is what that code looks like:

```
    'open a file, and print some information into it, then close it
    open "numbers.txt" for output as #out
    for x = 1 to 10
        print #out, x
    next x
    close #out

    'open the original file for input
    open "numbers.txt" for input as #in

    'open the new file for output
    open "numbers2.txt" for output as #out

[copyLoop]  'copy the contents of numbers.txt to numbers2.txt
    if eof(#in) = -1 then [doneCopying]
    input #in, value
    print #out, value
    goto [copyLoop]
```

```
[doneCopying]  'close numbers.txt & print new numbers into numbers2.txt
    close #in
    for x = 11 to 20
        print #out, x
    next x
    close #out

    'delete the original numbers.txt file
    kill "numbers.txt"
    'rename numbers2.txt to numbers.txt
    name "numbers2.txt" as "numbers.txt"

    end
```

Once the new file has been created containing the contents of the original file, plus our new numbers 11 to 20, we delete the old file using the KILL statement and rename the new file using the NAME statement.

There is a better way. Using the APPEND file mode, we can eliminate a lot of program code. The APPEND mode opens a file for writing, just like the OUTPUT mode does. Instead of erasing the contents of the file opened and starting at the beginning, the APPEND mode opens a file and retains everything. All we need to do then is write to the file, and everything we write will be added to the end of the file. Here is the example above rewritten using the APPEND mode:

```
    'open a file, and print some information into it, then close it
    open "numbers.txt" for output as #out
    for x = 1 to 10
        print #out, x
    next x
    close #out

    'open the original file for append
    open "numbers.txt" for append as #out
    for x = 11 to 20
        print #out, x
    next x
    close #out

    end
```

This is much shorter and simpler than the last example. When we are done writing the numbers 11 to 20 there is no need to delete or rename files.

Comma delimited files

By default, the standard way of reading from a sequential file using the INPUT statement breaks up the data we read at each end of line, and at each comma. Look at these equivalent file contents:

File 1 contents:

```
1,2,3,4,5,6,7,8,9,10
```

File 2 contents:

```
1,2,3,4,5
```

```
6,7,8,9,10
```

File 3 contents:

```
1
2
3
4
5
6
7
8
9
10
```

All three of the above possible file contents would produce the same displayed output using a loop like the following:

```
[loop]  'display each item on its own line
    if eof(#in) = -1 then [stopLooping]
    input #in, item$  'read the next item
    print item$
    goto [loop]
```

Since the comma is used to separate items (in addition to the end of line), this kind of file format is often called 'comma delimited'. This is a preferred generic format often used by commercial application software (spreadsheets are a good example) as a data export option. This is so that people like you and me can write our own customized software to use that information.

What if we want to use commas in our information? As an example, we decide that we will build an application that logs responses to a direct mail campaign, and we want the name to be entered in a single field. The name will be entered last name first, with a comma separating the last from the first name (example: Doe, John).

Take a look at this hypothetical file entry:

```
Doe, John
123 Main Street
Scottsdale
AZ
01234
```

Now look at this code fragment:

```
    input #in, name$
    input #in, street$
    input #in, city$
    input #in, state$
    input #in, zip$
```

If we read the file entry with the above code, name$ would only contain up to the first comma. So name$ would contain the string "Doe", street$ would contain "John", city$ would contain "123 Main Street", and so on. This is
not what we want.

There is a special form of the INPUT statement that will read a whole line from a file, including commas.

It is the LINE INPUT statement. Here is the code fragment above rewritten using LINE INPUT:

```
    line input #in, name$
    line input #in, street$
    line input #in, city$
    line input #in, state$
    line input #in, zip$
```

Another example of where LINE INPUT is useful is reading of a text document, like this one. There are many commas in this document. Here is a short program that uses the FILEDIALOG statement to ask for a *.txt file to read. Then it opens the file and reads and displays it.

```
    'ask for a filename
    filedialog "Select a text file", "*.TXT", filename$
    'if filename$ = "" then the user canceled the file selection
    if filename$ = "" then end

    'open the file
    open filename$ for input as #in

[readLoop]  'read and display each line in the text file
    if eof(#in) = -1 then [stopLooping]
    line input #in, lineOfText$
    print lineOfText$
    goto [readLoop]

[stopLooping]  'close the file
    close #in

    end
```

Run this small program and try it with a file that you know has some commas in it. Then change the line containing the LINE INPUT statement to use a regular INPUT statement and run the program again on the same file so you can see how they function differently.

Here's the assignment

Create a program called CALLER.BAS. This program will give the user the following options:

1) Enter a phone call
2) Search by caller's name
3) Search by person called
3) Quit

Menu item 1 will ask the user for phone call information and append it as a new record to the end of a file named PHONELOG.TXT. The added record must include:

1) Caller's name
2) Name of person called
3) Date of the call
4) Time of the call
5) A one line description of the call's purpose
6) A phone # where the caller can be reached

When searching, the program should stop and display the six data items for each record that matches and ask if the user wants to keep searching or quit the search. The program will display a notice if no matches are found.

Possible enhancement to CALLER.BAS

It is possible to search for a partial match.   Look at the following code example:

```
text$ = "The quick brown fox jumped over the lazy dog."
searchFor$ = "FOX"  'change FOX to different values and re-run
print "Searching for:"
print "  "; searchFor$
print "In the following:"
print "  "; text$
a$ = upper$(text$)
b$ = upper$(searchFor$)
if instr(a$, b$) > 0 then print "Found!" else print "Not Found!"
```

The UPPER$() function eliminates any letter case differences.   The INSTR() function returns a 0 value if our uppercased searchFor$ is not found in our uppercased text, or it returns the position of searchFor$ if it is found.

OUT

OUT port, byte

Description:

This command sends a byte value to the specified machine I/O port.   The use of this command inside of Windows is considered to be only for those in the know.   Windows provides no method of ensuring that more than one application will not access any I/O port at a time, so use this command with care (you know who you are).

See also: INP( )

INP( )

returnedByte = INP(port)

Description: This function polls the specified machine I/O port for its byte value.

See also: OUT

Using hexadecimal values

Liberty BASIC will let you specify hexadecimal values in your programs.   This is especially useful when calling API functions or using third party DLLs where the values specified in the documentation are in hexadecimal (base 16).   To convert a hexadecimal value, use the HEXDEC( )function.   Here is an example:

```
    print hexdec( "FF" )
```

Back to Making API and DLL Calls

HEXDEC( )

HEXDEC("value")

Description:

Returns a numeric decimal from a hexadecimal expressed in a string

Usage:

```
print hexdec( "FF" )
```

or if preferred:

```
print hexdec( "&HFF")
```

Using negative numbers in API calls

If you need to pass a negative value into an API or DLL function call, you will need to do a little bit of conversion. The following techniques will create two's complement versions of negative values that you can pass when a short or long is required (not a ushort or ulong).

Computing a negative short (2 bytes)

To pass -50 as a short:

```
        valueToPass = 0 or -50
  or:
        valueToPass = hexdec("10000") - 50
```

Computing a negative long (4 bytes)

To pass -10 as a long:

```
    valueToPass = hexdec("100000000") - 50
```

Back to Making API and DLL Calls

SPACE$(n)

Description:

This function will a return a string of n space characters (ASCII 32).   It is useful when producing formatted output to file or printer.

Usage:

```
for x = 1 to 10
    print space$(x); "*"
next x
```

AND	boolean and bitwise AND operator

The AND operator is used in

OPEN "COMn:baud,parity,data stop{,options}" for random as #handle

Description:

The OPEN "COMn:" statement opens a serial communications port for reading and writing.   This feature uses Microsoft Windows' own built-in communications API, so if you have a multiport communications card and a WIndows driver to support that card, you should be able to use any port on the card.

The simplest form for this command is:

```
OPEN "COMn:baud,parity,data,stop" for random as #handle
```

Allowable choices for baud are:

75 110 150 300 600 1200 2400 1800 2400 4800 9600 19200 38400 57600 115200

Allowable choices are parity are:

N	No parity
E	Even parity
O	Odd parity
S	Space parity
M	Mark parity

Allowable choices for data are:

5	bits long
6	bits long
7	bits long
8	bits long

Allowable choices for stop are:

1	stop bit
2	stop bits

Additional optional parameters can be included after the baud, parity, data and stop information:

CSn	Set CTS timeout in milliseconds (default 1000 milliseconds)
DSn	Set DSR timeout in milliseconds (default 1000 milliseconds)
PE	Enable parity checking
RS	Disable detection of RTS (request to send)

Other defaults:

DTR detection is disabled
XON/XOFF is disabled
binary mode is the default

To set the in and out communications buffers (each port has its own), set the variable Com (notice the uppercase C) to the desired size before opening the port.   Changing the variable after opening a port does not affect the size of the buffers for that port while it is open.

```
    'set the size of the communications buffers (in and out) to 16K each
    Com = 16384
```

Usage Notes:

To open com port 2 at 9600 baud, 8 data bits, 1 stop bit, and no parity, use this line of code:

```
    open "com2:9600,n,8,1" for random as #commHandle
```

It is recommended that you set the the timeout on the DSR line to 0 so that your program doesn't just freeze when waiting for data to come in. To do this, we can add a ds0 (for DSR 0 timeout) as below. Notice we use a different communications speed in this example.

```
    open "com2:19200,n,8,1,ds0" for random as #commHandle
```

Remember that when a modem dials and connects to another modem, it negotiates a speed to connect at. In the case of 14400 speed modems, you need to specify 19200 as the connection speed and let the modems work it out between themselves during the connect. This is because 14400 is not a baud rate supported by Windows (and you'll find that QBASIC doesn't directly support 14400 baud either).

Once the port is open, sending data is accomplished by printing to the port, like so (ATZ resets modems that understand the Hayes command set):

```
    print #commHandle, "ATZ"
```

To read from the port you should first check to see if there is anything to read. This is accomplished in this fashion:

```
    numBytes = lof(#commHandle)
```

Then read the data using the input$() function.

```
    dataRead$ = input$(#commHandle, numBytes)
```

Putting the lof( ) and input$( ) functions together on one line, it looks like this:

```
    dataRead$ = input$(#commHandle, lof(#commHandle))
```

When you're all done, close the port like so:

```
    close #commHandle
```

Week 3 Homework Solution

```
    'CALLER.LOG
    'Here is the solution to our homework assignment for
    'week 3 of our Liberty BASIC course
    'Copyright 1996 Shoptalk Systems
    'All rights reserved

[menu]  'display menu options

    cls
    print "**Caller Log Program**"
    print
    print " 1) Enter a phone call"
    print " 2) Search by caller's name"
    print " 3) Search by person called"
    print " 4) Quit"
    print
    print "Choose an option from 1 to 4."
    input ">"; option

    if option < 1 or option > 4 then gosub [badOption]
    if option = 1 then gosub [enterAPhoneCall]
    if option = 2 then gosub [searchByCallersName]
    if option = 3 then gosub [searchByPersonCalled]
    if option = 4 then [quit]

    goto [menu]

[badOption]  'display a notice that a bad selection was made

    print
    beep
    print "Option "; option; " is unsupported."
    print "Press any key."
    dummyVar$ = input$(1)

    return

[enterAPhoneCall]  'accept a phone log entry from the user

    cls
    print "**Enter a Phone Call**"
    print

    input "Caller's name ?"; callersName$
    input "Name of person called ?"; personCalled$
    input "Date of call (press 'Enter' for "+date$()+") ?"; dateOfCall$
    if dateOfCall$ = "" then dateOfCall$ = date$()
    input "Time of call (press 'Enter' for "+time$()+") ?"; timeOfCall$
    if timeOfCall$ = "" then timeOfCall$ = time$()
    input "Purpose of call ?"; purposeOfCall$
    input "Phone # where the caller can be reached ?"; callersPhone$
```

```
[saveEditCancelLoop]  'give the user the option to save, edit or abort

    cls
    gosub [displayEntryInfo]
    print "Save, Edit, Cancel Entry (S/E/C)?";
    answer$ = input$(1)

    if answer$ = "S" or answer$ = "s" then gosub [saveEntry] : goto [menu]
    if answer$ = "E" or answer$ = "e" then gosub [editEntry]
    if answer$ = "C" or answer$ = "c" then [menu]

    goto [saveEditCancelLoop]


[displayEntryInfo]  'display call information

    print "            Caller's name : "; callersName$
    print "   Name of person called : "; personCalled$
    print "             Date of call : "; dateOfCall$
    print "             Time of call : "; timeOfCall$
    print "          Purpose of call : "; purposeOfCall$
    print "Caller can be reached at : "; callersPhone$

    return


[editEntry]  'edit call information

    cls
    print "**Edit Caller Entry**"
    print

    print "            Caller's name : "; callersName$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then callersName$ = newEntry$

    print "   Name of person called : "; personCalled$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then personCalled$ = newEntry$

    print "             Date of call : "; dateOfCall$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then dateOfCall$ = newEntry$

    print "             Time of call : "; timeOfCall$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then timeOfCall$ = newEntry$

    print "          Purpose of call : "; purposeOfCall$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then purposeOfCall$ = newEntry$

    print "Caller can be reached at : "; callersPhone$
    input "  Press Enter, or retype > "; newEntry$
    if newEntry$ <> "" then callersPhone$ = newEntry$

    return
```

```
[saveEntry]  'write the entry info to PHONELOG.TXT

    open "PHONELOG.TXT" for append as #out

    print #out, callersName$
    print #out, personCalled$
    print #out, dateOfCall$
    print #out, timeOfCall$
    print #out, purposeOfCall$
    print #out, callersPhone$

    close #out

    return


[searchByCallersName]  'look for a phone log entry by caller's name

    cls
    print "**Search by Caller's Name**"
    print
    print "Please enter a partial name to search by."
    input ">"; searchCaller$
    if searchCaller$ = "" then [menu]  'nothing entered, abort search

    searchCaller$ = upper$(searchCaller$) 'convert to uppercase for search

    open "PHONELOG.TXT" for input as #in

    foundFlag = 0
    quitFlag = 0
    if eof(#in) = -1 then [endOfCallerSearch]


[searchByCallerLoop]
    gosub [readEntry]  'get next entry from PHONELOG.TXT
    if instr(upper$(callersName$), searchCaller$) > 0 then gosub [matched]
    if eof(#in) = 0 and quitFlag = 0 then [searchByCallerLoop]


[endOfCallerSearch]
    close #in
    if foundFlag = 0 then print "No matches."
    print "Press any key."
    dummyVar$ = input$(1)
    goto [menu]


[searchByPersonCalled]  'look for a phone log entry by person called

    cls
    print "**Search by Person Called**"
    print
    print "Please enter a partial name to search by."
    input ">"; searchCalled$
    if searchCalled$ = "" then [menu]  'nothing entered, abort search

    searchCalled$ = upper$(searchCalled$) 'convert to uppercase for search

    open "PHONELOG.TXT" for input as #in

    foundFlag = 0
    quitFlag = 0
    if eof(#in) = -1 then [endOfCalledSearch]


[searchByCalledLoop]
    gosub [readEntry]  'get next entry from PHONELOG.TXT
    if instr(upper$(personCalled$), searchCalled$) > 0 then gosub [matched]
    if eof(#in) = 0 and quitFlag = 0 then [searchByCalledLoop]


[endOfCalledSearch]
    close #in
    if foundFlag = 0 then print "No matches."
    print "Press any key."
    dummyVar$ = input$(1)
    goto [menu]


[readEntry]  'read the next entry from PHONELOG.TXT

    line input #in, callersName$
    line input #in, personCalled$
    line input #in, dateOfCall$
    line input #in, timeOfCall$
    line input #in, purposeOfCall$
    line input #in, callersPhone$

    return


[matched]  'stop and show a match & ask what to do next

    foundFlag = 1
    print "---------Match---------"
    gosub [displayEntryInfo]
    print
    print "Next Entry, Quit Searching (N/Q)?"
    answer$ = input$(1)
    'only check for quit response
    if answer$ = "Q" or answer$ = "q" then quitFlag = 1

    return


[quit]  'end CALLER.BAS here

    end
```

Week Four - Programming the Windows Graphical User Interface

Week 4 course material
Liberty BASIC programming course

Programming the Windows Graphical User Interface

What is a GUI (graphical user interface) anyway? This probably means different things to different people but to most it means a way of controlling a computer that involves manipulating windows (rectangular areas displayed on a computer screen) using a mouse or other pointing device.

If you are using Microsoft WINDOWS then you are using a software system with a GUI (we will use all uppercase when referring to Microsoft's product and not to any kind of window in general). WINDOWS isn't just a GUI, but it does include one.

You have been using the WINDOWS GUI all along while working through this Liberty BASIC course. The programs that you've written each run in a window that is managed by the WINDOWS GUI. This simple text-only window is called a 'main window' in Liberty BASIC lingo. This part of the course is concerned with teaching how to do fancier things in the way of designing windows for your programs.

In this section we will learn:

1) How to use Liberty BASIC to open windows in the WINDOWS GUI;
2) How to set the size, position, and type of our windows;
3) How to add buttons and other controls to our windows;
4) How to program the buttons and other controls to do what we want

Note: All of the program examples in this section are also provided as individual program files.

Your First Window

Type in the following short program and run it. Insert your name in the place where it says myname.

```
'WK4PROG1.BAS
'open a window
open "myname's first window!" for window as #myFirst

'now stop and wait
input aVar$

'now close the window
close #myFirst

end
```

A window will appear with your name in the title! That was easy, wasn't it? Now click on the program's initial window (called its 'main window') and press Enter to respond to the INPUT statement. The window will close and the program will end.

Liberty BASIC uses file statements like OPEN, CLOSE, INPUT, and PRINT to manage and control windows. Once a window is opened, we can send it commands by PRINTing to it, and we can get information by INPUTting from it.

Note: There is nothing to prevent a Liberty BASIC program from opening more than one window as demonstrated. All you need is multiple OPEN statements, and each window must have a unique handle.

Adjusting Window Size and Position

Let's change the size of our Window. This is done by changing the value of two special variables WindowWidth and WindowHeight before opening the window. We can also determine the opening position of the window by adjusting two other special variables UpperLeftX and UpperLeftY. This positioning is relative to the upper-left corner of the display screen. Take a look at this example:

```
    'WK4PROG2.BAS
    'open a window sized at 300 by 100 at position 200, 150
    WindowWidth = 300
    WindowHeight = 100
    UpperLeftX = 200
    UpperLeftY = 150
    open "myname's first window!" for window as #myFirst

    'now stop and wait
    input aVar$

    'now close the window
    close #myFirst

    end
```

Adding a button

Now let's add a button to our window. Look at the following short program:

```
    'WK4PROG3.BAS
    'open a window with a button
    'the window is sized at 399 by 100 at position 200, 150
    WindowWidth = 300
    WindowHeight = 100
    UpperLeftX = 200
    UpperLeftY = 150
    button #myFirst.ok, "OK!", [okClicked], UL, 15, 15
    open "myname's first window!" for window as #myFirst

    'now stop and wait
    input aVar$
    goto [quit]

[okClicked]  'the OK! button was clicked

    notice "You clicked on OK!"

[quit]
```

```
'now close the window
close #myFirst

end
```

Notice the BUTTON statement we placed before the OPEN statement. This must be inserted BEFORE the OPEN statement. Let's break that line of code down to explain how it works.

```
button #myFirst.ok, "OK!", [okClicked], UL, 15, 15
```

The first item in the BUTTON statement is a handle for the button. In this case our handle is #myFirst.ok. We are adding this button to a window that will have a handle of #myFirst, and we add .ok to extend the handle so that the button will have its own handle different from the handle of the window.

The second item is the [okClicked] branch label. This tells Liberty BASIC to perform an implicit GOTO [okClicked] when the button is clicked on. In this case the NOTICE statement causes a dialog box to appear that displays "You clicked on OK!". Once you clear the dialog box the window will be closed and the program will end.

The third item (UL) tells Liberty BASIC that the button will be positioned relative to the upper-left corner of the window. There are also UR, LL, and LR (for placing buttons relative to the upper-right, lower-left and lower-right corners).

The fourth and fifth items specify how many pixels from the (in this case upper-left) corner to place the button.

One option is to specify the exact width and height of the button. Liberty BASIC automatically chooses the size of the button if you don't specify it. Here's the same button, but we'll change the size to 75 pixels wide and 50 pixels high:

```
button #myFirst.ok, "OK!", [okClicked], UL, 15, 15, 75, 50
```

The INPUT statement in our program is very important. Liberty BASIC will only respond to actions performed on buttons or other controls when a running program is waiting at an INPUT statement (or a SCAN statement, but we won't cover that here).

Adding an Entry Field with TEXTBOX

Now let's add a textbox control to our window that let's us display and edit a line of text. We will use the TEXTBOX statement to accomplish this. Look at this code:

```
'WK4PROG4.BAS
'open a window with a button and a textbox
'the window is sized at 300 by 100 at position 200, 150
WindowWidth = 300
WindowHeight = 100
UpperLeftX = 200
UpperLeftY = 150
button #myFirst.ok, "OK!", [okClicked], UL, 15, 15
textbox #myFirst.field, 60, 15, 200, 25
open "myname's first window!" for window as #myFirst

'now print some text into the textbox
print #myFirst.field, "Type some text here."
```

```
    'now stop and wait
    input aVar$
    goto [quit]

[okClicked]  'OK! was clicked.  Get the contents of the entry field

    'print a command to the textbox
    print #myFirst.field, "!contents?"
    'now get the contents from the textbox
    input #myFirst.field, aString$

    'now pop up a notice saying what was in the textbox
    notice aString$

[quit]
    'now close the window
    close #myFirst

    end
```

Run the example and see how it works! Notice the TEXTBOX statement after the BUTTON statement. Its format is very simple.

```
    textbox #myFirst.field, 60, 15, 200, 25
```

The first item #myFirst.field is a handle for the textbox. This is just like the handle we gave to our button.

The next two items 60 and 15 are the placement of the control relative to the upper-left corner of the window.

The last two items 200 and 25 are the width and height of the control. If it isn't wide enough, the end of the line will not appear. If it isn't tall enough, you won't see any text in the control at all.

After our OPEN statement, see how we print some text "Type some text here." into our textbox control. This is the text that you see in the control when the program is run.

Now look at how we changed the code after our [okClicked] branch label. Instead of just displaying the same message every time it displays the text in our textbox control. We accomplish this by printing a command to the textbox with the line:

```
    print #myFirst.field, "!contents?"
```

This tells Liberty BASIC that we want the contents of that textbox control. Then using the following line we get that string and insert it into the variable aString$:

```
    input #myFirst.field, aString$
```

When printing a command to a textbox control, the command must start with an exclamation point. This is how Liberty BASIC knows that we want to send a command to the textbox. The contents of our textbox would have been replaced with "contents?" if we had left out the exclamation point like so:

```
    print #myFirst.field, "contents?"
```

Labeling with Statictext Controls

So far our window designs have been simple enough that we haven't needed to mark any part of our window with a descriptive label or annotation. This is an important matter in many applications where more than one textbox (or other kind of control) is used. Here is a version of our program that uses a statictext control as a label:

```
    'WK4PROG5.BAS
    'open a window with a button, a textbox, and a statictext
    'the window is sized at 300 by 100 at position 200, 150
    WindowWidth = 300
    WindowHeight = 100
    UpperLeftX = 200
    UpperLeftY = 150
    button #myFirst.ok, "OK!", [okClicked], UL, 220, 35
    textbox #myFirst.field, 10, 35, 200, 25
    statictext #myFirst.label, "Type some text here.", 10, 10, 150, 25
    open "myname's first window!" for window as #myFirst

    'now stop and wait
    input aVar$
    goto [quit]

[okClicked]  'OK! was clicked.  Get the contents of the entry field

    'print a command to the textbox
    print #myFirst.field, "!contents?"
    'now get the contents from the textbox
    input #myFirst.field, aString$

    'now pop up a notice saying what was in the textbox
    notice aString$

[quit]
    'now close the window
    close #myFirst

    end
```

See how we move the button and the textbox around in the window, and we add a text description with instructions to the user. The format for the STATICTEXT statement is almost the same as for the TEXTBOX statement.

```
    statictext #myFirst.label, "Type some text here.", 10, 10, 150, 25
```

The first item is a string used for the text in our statictext control.

The second item #myFirst.field is a handle for the statictext. This is just like the handle we gave to our button.

The next two items 10 and 10 are the placement of the control relative to the upper-left corner of the window.

The last two items 150 and 25 are the width and height of the control. The size must be large enough to hold the text. If it isn't wide enough, the end of the line will not appear. If it isn't tall enough, you won't see any text in the control at all.

Trapping a Window Close Event

When running any of our examples above we can close our custom made window at any time by double-clicking on its menu box, by pressing Alt-F4, etc. Then WINDOWS will go ahead and close the window, but Liberty BASIC will still think it's open. This will cause some trouble if our code tries to perform some work with a window that just isn't there anymore.

The way around this problem is by sending the trapclose command to our window in question. The trapclose command tells Liberty BASIC to hook into the WINDOWS event that closes our window. Here is our program modified to use trapclose.

```
    'WK4PROG6.BAS
    'open a window with a button, a textbox, and a statictext
    'show how trapclose works
    'the window is sized at 300 by 100 at position 200, 150
    WindowWidth = 300
    WindowHeight = 100
    UpperLeftX = 200
    UpperLeftY = 150
    button #myFirst.ok, "OK!", [okClicked], UL, 220, 35
    textbox #myFirst.field, 10, 35, 200, 25
    statictext #myFirst.label, "Type some text here.", 10, 10, 150, 25
    open "myname's first window!" for window as #myFirst

    'send the trapclose command
    print #myFirst, "trapclose [quit]"

[waitHere]  'now stop and wait
    input aVar$
    goto [quit]

[okClicked]  'OK! was clicked.  Get the contents of the entry field

    'print a command to the textbox
    print #myFirst.field, "!contents?"
    'now get the contents from the textbox
    input #myFirst.field, aString$

    'now pop up a notice saying what was in the textbox
    notice aString$

[quit]
    'ask if the user wants to quit
    confirm "Really Quit?"; answer$
    if answer$ <> "yes" then [waitHere] 'abort quitting

    'now close the window
    close #myFirst

    end
```

See how the trapclose command that we print to our window also specifies the [quit] branch label. This means that when we try to close the window using a mouse or keyboard action (using the CLOSE statement doesn't trigger trapclose), then WINDOWS will not close the window. Instead Liberty BASIC will execute an implicit GOTO [quit].

Notice also how we added code following the [quit] branch label. Using a CONFIRM statement we bring up a dialog box to confirm that we really do want to quit.

We also added a [waitHere] branch label in from of our INPUT statement so that we have a common place for execution to stop and wait for more user interaction. This is pretty much standard practice in Liberty BASIC. It really doesn't matter at which INPUT statement a Liberty BASIC stops and waits for input, but it makes things simpler if we always know where things are happening.

Challenge Exercise

We can add many controls to a window in Liberty BASIC. If we need a dozen textboxes to hold different data items and if we need a statictext to label each textbox we can do it. Using WK4PROG6.BAS as a base, create a program that opens a window titled "Caller Record Data". This window contains several textbox controls, each labeled by a statictext control.

The labels for our textbox controls are from our Week 3 homework:

1) Caller's name
2) Name of person called
3) Date
4) Time of call
5) Purpose of call
6) Caller can be reached at

When the user clicks on OK!, the program should then extract all the information entered into the textbox controls and then close the window. The information should then be displayed in the program's main window.

Week 4 Homework

Week 4 Homework
Liberty BASIC programming course

Using NOMAINWIN

The default text window that opens with each Liberty BASIC program we create can be optionally turned off.   This can be desirable when we don't want our application to be operated out of a text window.   The statement that accomplishes this effect is NOMAINWIN.   Simply include NOMAINWIN in your program's source code anywhere at all.   This accomplishes two things:

1) It hides the program's main window;

2) It prevents the 'Execution Complete' notice from appearing at the end of a program run.

This comes at a cost.   Your program must close itself down properly when all the windows have been closed.   Below is a modified version of our WK4PROG6.BAS program.   In this case all we needed to do was add the NOMAINWIN statement.   No other changes are needed because we have an END statement after our example window and because we don't use the program's main window for anything.

```
    'WK4PROG7.BAS - (WK4PROG6.BAS modified)
    'open a window with a button, a textbox, and a statictext
    'show how trapclose works
    'the window is sized at 300 by 100 at position 200, 150

    nomainwin  'do not open a main window

    WindowWidth = 300
    WindowHeight = 100
    UpperLeftX = 200
    UpperLeftY = 150
    button #myFirst.ok, "OK!", [okClicked], UL, 220, 35
    textbox #myFirst.field, 10, 35, 200, 25
    statictext #myFirst.label, "Type some text here.", 10, 10, 150, 25
    open "myname's first window!" for window as #myFirst

    'send the trapclose command
    print #myFirst, "trapclose [quit]"

[waitHere]  'now stop and wait
    input aVar$
    goto [quit]

[okClicked]  'OK! was clicked.  Get the contents of the entry field

    'print a command to the textbox
    print #myFirst.field, "!contents?"
    'now get the contents from the textbox
    input #myFirst.field, aString$
    'now pop up a notice saying what was in the textbox
    notice aString$

[quit]
    'ask if the user wants to quit
    confirm "Really Quit?"; answer$
    if answer$ <> "yes" then [waitHere] 'abort quitting

    'now close the window
    close #myFirst

    end
```

If there was a GOTO [waitHere] statement after the CLOSE #myFirst statement in our program above, then our program would not terminate properly, and there would be no error message to clue us in.

The Structure of a Liberty BASIC GUI Program

We've taken a look at some simple elements of Liberty BASIC GUI programming. Now let's consider how to organize code in a logical and practical way when creating GUI programs in LB.

Organization

In the case of a project that uses more than one window, we should make sure that code for each window is grouped together. Most applications are designed around a cental window that is always open and from where other windows are launched. When the central window is closed, the application ends. Here is a simple outline (not a working program) that demonstrates this idea:

```
    'MYPROG.BAS
    'This is an example outline
    'A description of the program should go here

    '------------------------Let's define our main application window here
    button #main.1, ....
    button #main.2, ....
    textbox ....
    open "My Application" for window as #main
    print #main, "trapclose [closeMain]"


[waitWinMain]  'wait here for user action
    input aVar$
    goto [waitWinMain]


[button1Clicked]  'here is a routine that handles when a button is clicked
    'do something here
    'now go back and wait
    goto [waitWinMain]

[button2Clicked]  'here is a routine that handles when a button is clicked
```

```
    'do something here
    'now go back and wait
    goto [waitWinMain]


[closeMain]  'perform some cleanup code and close the window
    close #main
    gosub [sub1]  'act on user action

    'end application
    end



[openWin2]  '--------------------------Let's define our second window here

    button #2.3, ....
    button #2.4, ....
    textbox ....
    open "Window 2" for window as #2
    print #2, "trapclose [close2]"


[waitWin2]  'wait here for user action
    input aVar$
    goto [waitWin2]


[button3Clicked]  'here is a routine that handles when a button is clicked
    'do something here
    'now go back and wait
    goto [waitWin2]


[button4Clicked]  'here is a routine that handles when a button is clicked
    'do something here
    'now go back and wait
    goto [waitWin2]


[close2]  'perform some cleanup code and close the window
    close #2
    gosub [sub2]  'act on user action
    goto [waitWin2]


'----------------------------------------------------------------------------

    'We might place code here containing general subroutines
```

See how we arrange the code for each window like so:

[openMainWindow]
[userAction1]
[userAction2]

```
[closeMainWindow]
      end

[openWindow1]
[userAction3]
[userAction4]
[closeWindow1]

[openWindow2]
[userAction5]
[userAction6]
[closeWindow2]
```

The names we give to our branch labels above are of course generic. All the code for a given window and user actions performed on it should be grouped together. By structuring things in this way it is relatively easy to develop and debug your programs.

NOTE: Be careful to avoid creating a program that closes all its windows but doesn't actually end. If the program stops at an input statement after closing all of it's windows, it will not terminate properly and resources will not be freed. If the program loops continuously after closing all its windows then our WINDOWS message handler (the one set up to Liberty BASIC) will be locked out. In this case press CTRL-Break to halt the errant program.

Using Dialog Boxes

A dialog box is a special kind of window. They are different in several respects from other windows:

1) Pressing the Tab key causes WINDOWS to cycle through the controls added to a dialog box. Each control is made the active one in turn.

2) Dialog boxes can be made modal. This means that if I have a first window open and then I open a modal dialog box, that I will be unable to select the first window until the modal dialog is closed. Most 'About' boxes are modal dialogs, as are many 'Properties' type windows. For an example, select Liberty BASIC's Help menu and click on 'About Liberty BASIC'. You won't be able to do anything else with Liberty BASIC until you close the dialog box that appears.

3) In Liberty BASIC, dialogs ignore the UpperLeftX and UpperLeftY variables, so a dialog box's initial screen position cannot be determined in this way.

4) Dialog boxes cannot be resized by the user.

Some applications are appropriately constructed completely of dialog boxes and have no other kind of window at all. Take a look at the included program FreeForm (FF12.BAS). This program is more typical. The main application windows is a graphics type, but most of the other windows are dialog boxes.

'Canned' Dialog Boxes

There are some simple preconstructed dialog boxes in Liberty BASIC. These are useful for displaying simple notices, asking for a yes and no response, prompting the user to type some string, or selecting a filename from a list of filenames.

Here are some examples:

```
    'display a notice
    notice "File Not Found!"

    'ask a yes or no question
    confirm "Would you like to skip the tutorial?"; answer$

    'prompt for input string
    prompt "Please enter your name:"; name$

    'get a filename from the user
    filedialog "Pick a text file", "*.txt", selection$
```

There are right and wrong times to use these kinds of dialogs. Here is the HILO.BAS program written using only the above dialog boxes. This illustrates a wrong way to write a WINDOWS program.

```
    ' WK4PROG8.BAS
    ' Here is an interactive HI-LO
    ' Program

    'don't use a main window
    nomainwin

[start]
    guessMe = int(rnd(1)*100) + 1

    ' Clear the screen and print the title and instructions
    notice "HI-LO" + chr$(13) + _
     "I have decided on a number between one " + _
     "and a hundred, and I want you to guess " + _
     "what it is.  I will tell you to guess " + _
     "higher or lower, and we'll count up " + _
     "the number of guesses you use."

[ask]
    ' Ask the user to guess the number and tally the guess
    prompt "OK.  What is your guess ?"; guess$
    guess = val(guess$)

    ' Now add one to the count variable to count the guesses
    let count = count + 1

    ' check to see if the guess is right
    if guess = guessMe then goto [win]
    ' check to see if the guess is too low
    if guess < guessMe then notice "Guess higher."
    ' check to see if the guess is too high
    if guess > guessMe then notice "Guess lower."

    ' go back and ask again
    goto [ask]

[win]
    ' beep once and tell how many guesses it took to win
    beep
    notice "You win!  It took" + str$(count) + "guesses."

    ' reset the count variable to zero for the next game
    let count = 0

    ' ask to play again
    confirm "Play again (Y/N)?"; play$
    if instr("YESyes", play$) > 0 then goto [start]

    end
```

What is wrong with the program above? Technically there is nothing wrong with it because it works. However one obvious thing wrong is that there's only one place where you can exit the program (at the end of each game). It also looks sloppy. It doesn't have one application window that is open all the time as most WINDOWS programs do, so it will seem strange to most users.

More About Statictext

We saw in our earlier installment of this Week 4 lesson how a statictext control is used to label other controls in a window. Statictext controls do not have to always display the same label. We can print to them in similar fashion to the way we print to a textbox control. Try the example code below:

```
    'WK4PROG9.BAS
    'demonstrate printing to statictext control

    statictext #countdown.stext, "Counting down:", 10, 10, 200, 25
    open "Countdown" for window as #countdown

    for count = 10 to 1 step -1
        print #countdown.stext, "Counting down: "; count
        t$ = time$()
        while t$ = time$()
        wend
    next count

    print #countdown.stext, "Counting Down: Done."

    end
```

Homework Assignment

Create a version of the WK4PROG8.BAS program above (call it MYHILO.BAS). The program should have its own application window using the dialog window type. This window should have at least one statictext control. One of these should indicate whether to guess higher or lower and to count the guesses. There should be a textbox for entering guesses, and a button labeled guess to click on to register each guess. An additional button should be labeled 'About'. This button should cause an About box to be displayed for the game. The About box can be a canned dialog box, or you can construct one yourself using a modal dialog box.

Week 4 Homework Solution

```
    'MYHILO.BAS  -  Week 4 homework solution, Liberty BASIC course

    'This is a minimalistic program but it demonstrates the basics.
    'You may recognize some of this code from our example program,
    'but most of it is new.  Because we are using a dialog box, we
    'can tab between controls, or we can use the mouse.

    'Don't use a main window
    nomainwin

    'Set the width and height of our dialog box
    WindowWidth = 312
    WindowHeight = 145

    'Set up our controls
    statictext #myhilo.instruct, "Enter your guess here:", 14, 16, 176, 20
    textbox #myhilo.guessField, 14, 41, 216, 25
    button #myhilo.guessNow, "Guess", [guessNow], UL, 238, 41, 50, 25
    statictext #myhilo.status, "Status Line:", 14, 81, 176, 20
    button #myhilo.guessNow, "About", [aboutMyhilo], UL, 238, 81, 50, 25

    'Open our program's dialog box
    open "Myhilo Game -  WK4SOL" for dialog as #myhilo

    'When the user want to close our window, goto [quit]
    print #myhilo, "trapclose [quit]"

    'Let's display our about-box
    gosub [aboutMyhiloSub]


[startGame] 'Start a new game of MYHILO

    guessMe = int(rnd(1)*100) + 1
    print #myhilo.guessField, "-Ready for your guess-"


[myhilo.inputLoop]   'Wait here for input event
    input aVar$
    goto [myhilo.inputLoop]


[guessNow]   'Perform action for the button named 'guessNow'

    'Get the guess from our guessField
    print #myhilo.guessField, "!contents?"
    input #myhilo.guessField, guess

    'Now add one to the count variable to count the guesses
    let count = count + 1

    'Check to see if the guess is right
    if guess = guessMe then [win]

    'Check to see if the guess is too low
    if guess < guessMe then status$ = "Guess higher."

    'Check to see if the guess is too high
    if guess > guessMe then status$ = "Guess lower."

    print #myhilo.status, status$; " "; count; " guesses."

    'Go back and wait for more input
    goto [myhilo.inputLoop]


[win]
    'Beep once and tell how many guesses it took to win
    beep
    print #myhilo.status, "You won in " ; count ; "! Play again."

    'Reset the count variable to zero for the next game
    count = 0

    goto [startGame]


[aboutMyhilo]  'call our about-box subroutine

    gosub [aboutMyhiloSub]
    goto [myhilo.inputLoop]


[aboutMyhiloSub]  'display information about our program

    'We could have used a NOTICE statement here, but I wanted to show
    'how to do this using a modal dialog window type.

    'Set the size of our about box
    WindowWidth = 368
    WindowHeight = 190

    'Create statictext controls and an OK button to close the about box
    statictext #about.stext1, "I have decided on a number between one", 22,
16, 312, 20
    statictext #about.stext2, "and a hundred, and I want you to guess", 22,
36, 320, 20
    statictext #about.stext3, "what it is.  I will tell you to guess", 22,
56, 312, 20
    statictext #about.stext4, "higher or lower, and we'll count up", 22, 76,
296, 20
    statictext #about.stext5, "the number of guesses you use.", 22, 96, 256,
20
    button #about.OK, "OK", [closeAboutBox], UL, 278, 111, 64, 35

    'Open our about dialog box as modal
    open "About Myhilo" for dialog_modal as #about

    'Use the same close code as the OK button
    print #about, "trapclose [closeAboutBox]"

    return


[closeAboutBox]   'Perform action for the button named 'OK'

    close #about
    goto [myhilo.inputLoop]


[quit] 'Quit our MYHILO program

    close #myhilo
    end
```

Week 5 - Introduction to Programming Graphics

An Introduction to Programming Graphics

One of the most interesting and useful things that can computers can do is create graphics. These can be useful diagrams, page layouts, animations, beautiful works of art, etc.

Kinds of Graphics Controls

Liberty BASIC programs can display graphics using two different types of graphics controls. The first is a graphics window. We create a graphics window in our programs using an OPEN statement in the same fashion we create other kinds of windows:

```
open "My Drawing Area" for graphics as #graphWin
```

We can also add one or more graphicbox controls to other windows, like so:

```
graphicbox #main.graph, 10, 10, 150, 200
open "My Window" for window as #main
```

Both kinds of graphics controls understand the same set of graphics commands. Using the second approach can provide more than one drawing area, or it can be used to create a graphics area with other controls in a window or dialog box.

Turtle Graphics

In many computer graphics systems there is a way to draw using a pen and paper metaphor. Because some versions of this system used real paper, and also a pen attached to a small robot that looked like a turtle (a hemisphere), this way of drawing is often referred to as turtle graphics.

The idea is that you send the turtle commands like:

-raise the pen up from the paper
-lower the pen down to the paper
-move forward a specified distance, drawing if the pen is down
-turn a specific number of degrees

By stringing these kinds of drawing commands together, useful and meaningful things can be drawn.

Liberty BASIC includes support for this kind of drawing. Here is a short list of the Liberty BASIC turtle graphics commands:

up	- lift the pen up (don't draw)
down	- lower the pen down (draw)
home	- place the pen in the center of the graphics area

go   - move foreward in the current direction
goto   - go to a specified position
place - go to a specified position but don't draw
turn   - turn the turtle clockwise a specified number of degrees
north - cause the turtle to point north (straight up)
posxy - tell us what the location of the turtle is

Let's draw a box in the center of a graphics control:

```
'draw a box
open "Draw A Box" for graphics as #draw
print #draw, "up"
print #draw, "home"
print #draw, "north"
print #draw, "go 50"
print #draw, "turn 90"
print #draw, "go 50"
print #draw, "down"
print #draw, "turn 90"
print #draw, "go 100"
print #draw, "turn 90"
print #draw, "go 100"
print #draw, "turn 90"
print #draw, "go 100"
print #draw, "turn 90"
print #draw, "go 100"
input r$
```

Run the program and look at the box we've drawn.   Now maximize the window.   See how the box doesn't get redrawn?   This is because we haven't told Liberty BASIC that we're done drawing a sequence.   To make what we've drawn 'stick', we must send a FLUSH command.   Insert the following line before our INPUT statement:

```
print #draw, "flush"
```

Now rerun the program and maximize the window.   You will see that the box is redrawn now whenever the window is resized or redrawn.

Doing more with less

For what our box drawing program does it is quite long.   There are a couple of things we can do to shorten the program.   One is to print more than one command on a line.   Look at this:

```
print #draw, "up"
print #draw, "home"
print #draw, "north"
print #draw, "go 50"
print #draw, "turn 90"
print #draw, "go 50"
print #draw, "down"
```

An equivalent line is:

```
print #draw, "up ; home ; north ; go 50 ; turn 90 ; go 50 ; down"
```

See how we can print more than one command per line by inserting a semicolon in between each command.

Now we turn the turtle 90 degrees and go 100 pixels.   This happens four times:

```
    print #draw, "turn 90"
    print #draw, "go 100"
    print #draw, "turn 90"
    print #draw, "go 100"
    print #draw, "turn 90"
    print #draw, "go 100"
    print #draw, "turn 90"
    print #draw, "go 100"
```

With this example we can combine the TURN and GO commands into one line with a semicolon, and we can use a loop to perform the action four times, like so:

```
    for x = 1 to 4
        print #draw, "turn 90 ; go 100"
    next x
```

Here is the complete program rewritten as above:

```
    'draw a box
    open "Draw A Box" for graphics as #draw
    print #draw, "up ; home ; north ; go 50 ; turn 90 ; go 50 ; down"
    for x = 1 to 4
        print #draw, "turn 90 ; go 100"
    next x
    print #draw, "flush"

    input r$
```

For fun, here is a short program that draws a spiral:

```
    'draw a spiral
    open "Draw A Spiral" for graphics as #draw
    print #draw, " home ; down"
    for x = 1 to 100
        print #draw, "turn 91 ; go "; x * 2
    next x
    print #draw, "flush"

    input r$
```

See how we take the value of x in our FOR/NEXT loop and use it in our GO command.   We do this to make each next line longer than the last, and this is what makes our spiral get bigger as we draw it.

IMPORTANT: Spaces are important in any command that we print to a graphics control.   If we remove the spaces, the command will not work, and it will not return an error.   For example, the following line will NOT work:

```
        print #draw, "turn 91 ; go"; x * 2
```

See how the space was removed after the GO command?   This small change is very important.

More about FLUSH

FLUSH doesn't only make drawn items 'stick' so that they redraw. It gives all the items drawn before that FLUSH a common number. This is useful so that a collection of drawing elements can be referred to together.

Here is an example of how we might use this feature:

```
'demonstrate the usefulness of flush
open "Draw A Circle" for graphics as #draw
print #draw, " home ; north ; down"
for x = 1 to 30
    print #draw, "turn 12 ; go 10"
    print #draw, "flush"
next x

for x = 1 to 29 step 2
    print #draw, "delsegment "; x
next x

print #draw, "redraw"

input r$
```

Notice how we flush after each TURN and GO command. This give each line its own segment number. Then we loop through the odd numbers 1, 3, 5, ... 29 so we can delete every other segment. When we print the REDRAW command, it only redraws the segments we haven't deleted and we see a dashed effect.

This segmented graphics capability is used a lot in the FreeForm program (FF12.BAS) that is included with Liberty BASIC.

We can get the current segment number at any time while drawing by using the SEGMENT command, like so:

```
'demonstrate the usefulness of flush
open "Draw A Circle" for graphics as #draw
print #draw, " home ; north ; down"
for x = 1 to 30
    print #draw, "turn 12 ; go 10"
    print #draw, "flush"
    'get the current segment number and print it
    print #draw, "segment"
    input #draw, segId
    print segId
next x

for x = 1 to 29 step 2
    print #draw, "delsegment "; x
next x

print #draw, "redraw"

input r$
```

The SEGMENT command does NOT get the number of the last segment flushed. Subtract 1 to get that number.

Color Graphics

We can dress up our graphics by adding some color to them. Let's look at two commands for doing this:

```
'draw a spiral
open "Draw A Spiral" for graphics as #draw
print #draw, "fill blue"
print #draw, " home ; down ; color white"
for x = 1 to 100
    print #draw, "turn 91 ; go "; x * 2
next x
print #draw, "flush"

input r$
```

The FILL command (the line after our OPEN statement) causes our graphic control to be filled with the color blue. Then our spiral is drawn as white by setting the turtle's pen to white in the next line.

Users running 65,000 or more color drivers on their machine can specify RGB values for colors. Each color can be given a value from 0 to 255 with 0 being darkest and 255 being lightest. Try this example:

```
'draw a spiral
open "Draw A Spiral" for graphics as #draw
print #draw, "fill darkgray"
print #draw, " home ; down"
for x = 1 to 100
    print #draw, "color "; x * 2 + 55; " 0 " ; 200 - x
    print #draw, "turn 91 ; go "; x * 2
next x
print #draw, "flush"

input r$
```

Here is a list of valid colors:

black, blue, brown, cyan, darkblue, darkcyan, darkgray,
darkgreen, darkpink, darkred, green, lightgray, palegray,
pink, red, white, yellow

Line Thickness

So far everything we've drawn has been razor-thin. Drawing with a bold line is simple matter. The SIZE command is all that's needed. Try this:

```
'draw a box
open "Draw A Box" for graphics as #draw
print #draw, "up ; home ; north ; go 50 ; turn 90 ; go 50 ; down"
'use a big fat pen
print #draw, "size 10"
```

```
    for x = 1 to 4
        print #draw, "turn 90 ; go 100"
    next x
    print #draw, "flush"

    input r$
```

If no size is specified, the default is 1.

Drawing Text with Graphics

Drawing text in a graphics control is simple. Position the pen as desired and print the text with an backslash in front. Take a look:

```
    'display some text
    open "Text example" for graphics as #draw
    print #draw, "place 50 50"
    print #draw, "\Hello world!"
    print #draw, "flush"

    input r$
```

Each backslash in printed text causes a new line to be printed. So if you want to display text that has a backslash in it (like a file path), then begin your text with a vertical bar '|' like so:

```
    'display some text
    open "Text example" for graphics as #draw
    print #draw, "place 50 50"
    print #draw, "|c:\config.sys"
    print #draw, "flush"

    input r$
```

You can use the COLOR command to change the color of printed text. Watch what happens when you also fill the window with some other color.

```
    'display some text
    open "Text example" for graphics as #draw
    print #draw, "fill red ; color blue"
    print #draw, "place 50 50"
    print #draw, "|c:\config.sys"
    print #draw, "flush"

    input r$
```

To change this white backdrop to the be the same color as our FILL of red, we can use the BACKCOLOR command, like this:

```
    'display some text
    open "Text example" for graphics as #draw
    print #draw, "fill red ; color blue ; backcolor red"
    print #draw, "place 50 50"
    print #draw, "|c:\config.sys"
    print #draw, "flush"
    input r$
```

Challenge Exercise

Create a program named TIME.BAS that gets the current time from the TIME$( ) function and then draws a clock with an hour, minute and second hand. The display should have a background color, and each hand should have its own color. A button on the clock should cause the display to redraw with the current time when clicked on. The program doesn't have to run continuously.

Week 5 Homework

Week 5 Homework Assignment
Liberty BASIC programming course

More on Programming Graphics

In the first part of week 5 we covered turtle graphics, drawing segments, drawing with color, and displaying text.   Now we will examine the use of bitmaps in Liberty BASIC.

NOTICE: This week's homework includes four bitmap files named LETTER.BMP, CALL.BMP, BROCHURE.BMP, and SALE.BMP for use in your assignment.

Using Bitmaps

In WINDOWS, a bitmap is a rectangular area containing an image.   WINDOWS supports bitmaps made up of just two colors (black & white), bitmaps made up of millions of colors, and everything in between.   Anyone can create their own bitmaps using WINDOWS Paintbrush and save them as *.BMP files.   There are other ways to create bitmap files, including the use of scanning equipment to produce an image from paper.   Electronic cameras are also becoming very popular.

Here we will examine Liberty BASIC's commands for loading bitmaps from a disk file and for displaying bitmaps into a graphics window or graphicbox control.

Here is a short program that loads one of the bitmaps that comes with Liberty BASIC and displays it in a graphics window:

```
'open a graphics window
open "Display a bitmap" for graphics as #bitmap

'load a bmp file, calling it "aBitmap"
loadbmp "aBitmap", "bmp\titlettt.bmp"

'draw the bitmap named aBitmap at position 50, 50 and flush it
print #bitmap, "drawbmp aBitmap 50 50"
print #bitmap, "flush"

input r$
```

The LOADBMP statement is used to load the bitmap "bmp\titlettt.bmp" into memory.   Liberty BASIC gives this bitmap the name "aBitmap" ("aBitmap" is NOT the filename of the bitmap file).   See how this is used when sending the DRAWBMP command to the graphics window.

We can combine drawn graphics and bitmaps.   Here is a small program that combines a drawn spiral and a bitmap file.

```
'combine graphics
open "Draw combined graphics" for graphics as #draw

'draw a spiral
print #draw, " home ; down"
for x = 1 to 100
```

```
        print #draw, "turn 91 ; go "; x * 2
    next x

    'load a bmp file, calling it "aBitmap"
    loadbmp "aBitmap", "bmp\titlettt.bmp"

    'display the bitmap named aBitmap at position 100, 100
    print #draw, "drawbmp aBitmap 100 100"

    'flush the combined drawing
    print #draw, "flush"

    input r$
```

Here's an example of this program that draws in a graphicbox instead of a graphics window type:

```
    'draw in a graphicbox
    graphicbox #main.draw, 10, 10, 150, 150
    open "Graphicbox example" for dialog as #main

    'draw a spiral
    print #main.draw, " home ; down"
    for x = 1 to 100
        print #main.draw, "turn 91 ; go "; x * 2
    next x

    'load a bmp file, calling it "aBitmap"
    loadbmp "aBitmap", "bmp\titlettt.bmp"

    'display the bitmap named aBitmap at position 100, 100
    print #main.draw, "drawbmp aBitmap 25 45"

    'flush the combined drawing
    print #main.draw, "flush"

    input r$
```

Using the idea above, a drawn graphic doesn't have to dominate the entire window.

Homework assignment

Write a program called STATUS.BAS that displays a graphic for one of these four status items:

- initial letter
- phone call
- mail information pack
- sale made

A dialog box should contain a graphicbox control to display the graphic, a listbox control to contain a list of the status item, and a statictext control to instruct the user to select a status item.

Four bitmap files have been included with this week's homework assignment. Their names are LETTER.BMP, CALL.BMP, BROCHURE.BMP, and SALE.BMP. You may use these files to display the graphic for each status item. Here is a short program that shows how to use a listbox control:

This let's take this idea and add some other controls to the dialog box:

```
    'setup some colors
    dim color$(5)
    color$(0) = "red"
    color$(1) = "green"
    color$(2) = "blue"

    statictext #main, "Pick a color:", 10, 10, 120, 20
    listbox #main.colors, color$(, [loop], 10, 35, 120, 60
    open "Listbox example" for dialog as #main

    'tell the listbox where to branch on a single-click
    print #main.colors, "singleclickselect [colorSelected]"


[loop]  'wait here for user input
    input r$
    goto [loop]

[colorSelected]  'draw spiral in the selected color

    print #main.colors, "selection?"
    input #main.colors, selectedColor$

    notice "You selected " + selectedColor$

    goto [loop]
```

Week 5 Homework Solution

```
    'WK5SOL.BAS
    'Week 5 Liberty BASIC Course
    'Copyright 1996 Shoptalk Systems
    'All Rights Reserved
    'Here is our solution for week 5 of our Liberty BASIC course
    '-------------------------------------------------------------------

    'load our bitmaps
    loadbmp "letter", "letter.bmp"
    loadbmp "call", "call.bmp"
    loadbmp "brochure", "brochure.bmp"
    loadbmp "sale", "sale.bmp"

    'setup our listbox choices
    dim status$(4)
    status$(0) = "letter"
    status$(1) = "call"
    status$(2) = "brochure"
    status$(3) = "sale"

    'no main window
    nomainwin

    'size the window
    WindowWidth = 200
    WindowHeight = 150

    'setup our controls and open the window
    statictext #main, "Select Status:", 10, 10, 120, 20
    listbox #main.status, status$(, [loop], 10, 35, 120, 60
    graphicbox #main.draw, 140, 35, 34, 34
    open "Week 5 Homework" for dialog as #main
    print #main, "trapclose [quit]"

    'tell the listbox where to branch on a single-click
    print #main.status, "singleclickselect [statusSelected]"


[loop]  'wait here for user input
    input r$
    goto [loop]


[statusSelected]  'draw the icon for the selected status

    print #main.status, "selection?"
    input #main.status, selectedStatus$

    print #main.draw, "cls ; drawbmp "; selectedStatus$; " 0 0"

    goto [loop]
```

```
[quit]  'exit the program

    close #main
    end
```

Week 6 - Building Our Application

Week 6 course material
Liberty BASIC programming course

Files included with this lesson

`WK6CM1.TXT`	– This file
`TRACKER.BAS`	– The finished solution to our week 6 project
`LETTER.BMP` `CALL.BMP` `BROCHURE.BMP` `SALE.BMP` `DEADEND.BMP`	– These five files are used as icons in our application
`LETTER.TXT`	– An example letter of introduction for our application
`PACKSLIP.TXT`	– An example packing slip for our application

Building Our Application

We now arrive at the final stage of our Liberty BASIC course. We will build a kind of personal information manager for managing sales calls. Our application will manage an address list. Each sales lead in the list will have a field containing an event status for our sales effort. Here is a list of events for each sales lead:

- Letter of Introduction
- Follow-Up Phone Call
- Brochure
- Sale
- Dead End Lead

Our application will provide a means of printing the Letter of Introduction, personalized with each name and also dated. We will launch NOTEPAD.EXE to edit our form letter, and we will used tags (like <nameTag> and <dateTag>) to insert the name and date where desired. See the included example file LETTER.TXT. When we have printed a Letter of Introduction for any record, its event status will automatically advance to Follow-Up Phone Call.

When making our Follow-Up Phone Call, our application will let us specify if there was:

- no contact, try later
- positive contact, send brochure
- negative contact, dead end lead

When we have had positive contact, the event status of the record will be advanced to brochure.

When the sale is made, we will print a packing slip using a technique similar to the one we will use for printing our letter of introduction. See the included example file PACKSLIP.TXT.

Here is a functional description of our application:

1) Main window - When the program is started, a window will open with the following controls:

- a) A combobox with a list of events (letter, call, brochure, etc). When we select an event in this control, our application code will fill our listbox below c) with leads that have the selected event status.

- b) A graphicbox for displaying an icon that represents the selection in our combobox above a). This is similar to our week 5 homework.

- c) A listbox for holding the names of sales leads. We will want to select a name from this list to perform actions on.

- d) A statictext to describe what possible action we can take on a lead we select in our listbox above c).

- e) A button to invoke the action described in our statictext d).

So if we want to work on making phone calls, we select 'call' from the combobox a). The graphicbox b) displays our call.bmp file, and listbox c) fills up with leads that have an event status of 'call'. The statictext d) displays 'Make Call'. Then we select a lead from the listbox c) and click on button e) and another window opens to present the information we need to make the call. Assuming the call is a success, we provide a way to register this in our application, and the event status of this sales lead record is advanced to 'brochure'.

Our main window will also need buttons for the following:

- a) New Lead - Open a window for entering a new record. Each record has these fields:

  - First Name
  - Last Name
  - Address 1
  - Address 2
  - City
  - State
  - Zip
  - Phone #
  - Comment
  - Event Status (not user editable)
  - Date of Last Action (not user editable)

- b) Edit/View Lead - Open a window to view or edit a record. The fields are the same as above.

c) Edit Letter - Run NOTEPAD.EXE to edit a letter of introduction

d) Edit Packslip -Run NOTEPAD.EXE to edit a packing slip

Try out TRACKER.BAS

To get a good feel for how the application should work, try the included TRACKER.BAS program file. Notice how we use dialog boxes, most of them modal dialog boxes. See how we run NOTEPAD.EXE to edit our templates for our letter of introduction and our packing slip.

Because this is a large project, it might be helpful for you to take a good look at the TRACKER.BAS file. Running the debugger on it might also be a good idea. Then once you have a good idea how things work, start writing your own code.

File Input/Output

The sales lead records are loaded when the program is started into an array 500 by 12. This will work with 500 records (an arbitrary limit). The first line in our file is a count of the number of records in the file. Each record's fields are stored one to a line, and LINE INPUT is used to read them in so that commas can be included in a field's text. When we quit the program, the records are saved back to disk.

Listboxes

The listbox is similar to the combobox we used in our week 5 homework. It fills itself with the contents of an array named in its originating statement, like so:

```
listbox #main.list, names$(, 10, 10, 100, 250
```

In the statement above, the listbox will fill itself with the contents of they array named names$(), which must be single dimensioned. Whenever the contents of the listbox need to change, the RELOAD command is used, like so:

```
print #main.list, "reload"
```

Printing the letter of introduction and packing slip

Text is sent to the printer using the LPRINT statement. The following will send Hello World to the printer:

```
lprint "Hello World"
dump  'force the end of page
```

The DUMP statement forces the end of page, which is what you will want to do each time you print a letter of introduction or print a packing slip.

UpperLeftX and UpperLeftY

Description:

The special variables UpperLeftX and UpperLeftY specify the distance from the top-left of the display that the next opened window will be.   For example, the following will open a graphics window 50 pixels from the left of the display, and 25 pixels from the top of the display:

```
UpperLeftX = 50
UpperLeftY = 25
open "test window" for graphics as #testHandle

input r$
```

See also: WindowWidth and WindowHeight

WindowWidth and WindowHeight

Description:

The special variables WindowWidth and WindowHeight specify the width and height of the next window to be opened. If your program's code does not specify the values for these special variables, their defaults will be 320 and 360 respectively. The following will open a graphics window 250 pixels wide and 100 pixels high.

```
WindowWidth = 250
WindowHeight = 100
open "test window" for graphics as #testHandle

input r$
```

See also: UpperLeftX and UpperLeftY

CommandLine$

Description:

This special variable contains any switches that were added when Liberty BASIC was started.   This is especially useful in applications executing under the runtime engine.

DisplayWidth and DisplayHeight

Description:

The special variables DisplayWidth and DisplayHeight contain the width and height respectively of the display screen.