GUI Programming

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:

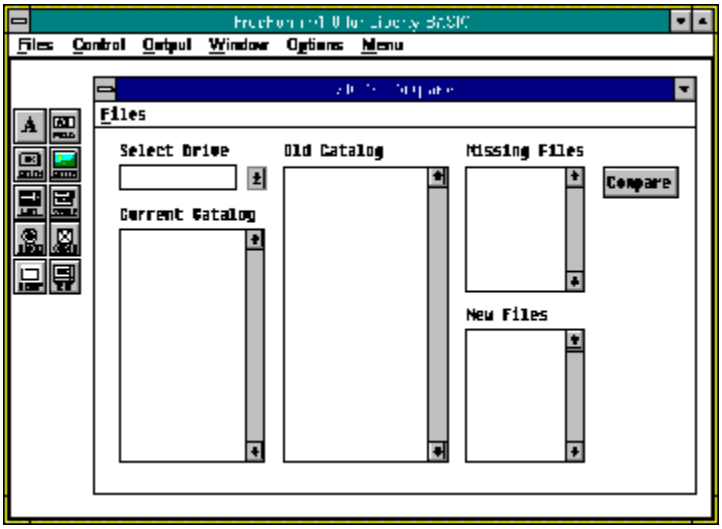
```
    button #graph, Exit, [exit], LR, 5, 5      'window will have a button
    open "Example" for graphics as #graph      'open graphics window
    print #graph, "up"                          'make sure pen is up
    print #graph, "home"                         'center the pen
    print #graph, "down"                         '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).

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

Size and Placement of Windows

The size and placement of any window can be easily determined before it is opened in Liberty BASIC (except for any window type with a _fs in its descriptor). 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

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.

Controls - Menus, Buttons, Etc.

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

Button

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

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

or

```
BUTTON #handle, "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 - This needs to be the same as the handle of the window.

"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.

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
```

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 do not work with windows of type dialog (this is bug in Liberty BASIC), but only with windows of type graphics or window. Instead of groupbox, try using statictext to label groups of controls. Make sure also that the controls you wish to contain with the groupbox are listed after the groupbox statement.

Graphics Window Commands

Most of these commands work only with windows of type graphics.

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

Or if you prefer:

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

will work just as well, and executes slightly faster.

Here are the commands:

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, "cls"

Clear the graphics window to white, erasing all drawn elements

print #handle, "fill COLOR"

Fill the window with COLOR. For a list of accepted colors see the color command below.

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, "down"

Just the opposite of up. This command reactivates the drawing

process.

print #handle, "home"

This command centers the pen in the graphics window.

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, "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, "goto X Y"

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

print #handle, "place X Y"

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

print #handle, "go D"

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

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, "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, "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, "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, "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, "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, "print"

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

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.

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, "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, "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, "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, "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, "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, "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, "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, "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, "rule rulename"

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

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.

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. 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

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.

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 of our text window is:"
print string$
input "Press 'Return'"; r$
close #1                               'close the window
```

Here are the text window 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.

```
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).

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.