The ABCs of APIs Lesson 1

Getting Started

Table of Contents

This is the first in a series of tutorials for using API calls. It covers the most fundamental information. Later tutorials will go into more depth.

What is an API Call?

An API call is a **function** contained in a library file, called a **Dynamic Link Library** or **DLL**. These files have an extension of "*.DLL".

Here is an example of an API call. We can use it to determine if a window or control is enabled or disabled.

```
CallDLL #user32, "IsWindowEnabled",_
   hButton as uLong,_   'handle of window or control
   result As Long       'nonzero = enabled
```

Liberty BASIC allows us to enable or disable a window or control, but there is no Liberty BASIC function to discover if a window or control is currently enabled or disabled. We can get around that limitation with this handy API function. Here is a Liberty BASIC program that makes use of the "IsWindowEnabled" API function. Copy it and paste it into Liberty BASIC, then run it to see how it works.

```
nomainwin

'open a program window
button #1.enable, "Enable/Disable",[doEnable],UL,10,10
button #1.quit, "Quit", [quit], UL, 10, 60
open "API Demo" for window as #1
#1 "trapclose [quit]"

'get the handle of the quit button
hButton = HWND(#1.quit)

wait

[doEnable]
'The IsWindowEnabled function determines whether the
'specified window or control is enabled
'for mouse and keyboard input

CallDLL #user32, "IsWindowEnabled",_
   hButton as uLong,_   'handle of window or control
   result As Long       'nonzero = enabled

'a result of 0 means that the window is NOT enabled
'a nonzero result means that the window IS enabled

if result = 0 then
    'if button was disabled, enable it
    #1.quit "!enable"
else
    'if bitton was enabled, disable it
    #1.quit "!disable"
end if
```

```
wait

[quit] close #1 :  end
```

Functions

What is a function? It is a block of code that takes in an argument or a list of arguments and returns a value. That sounds a little complicated, doesn't it? It's kind of like my bread machine. I put a list of ingredients into my bread machine, including flour, yeast and water. My bread machine takes these ingredients and manipulates them, then returns a loaf of fresh-baked bread to me.

The list of ingredients can change. If I put in white flour, my bread machine returns white bread. If I use rye flour, it returns rye bread. If I use whole wheat flour, it returns wheat bread.

A function accepts a list of ingredients that we call arguments. (They can also be called parameters.) It manipulates these arguments and returns a value. Liberty BASIC has many built-in functions. Some examples of native Liberty BASIC functions are **MAX()**, **MIN()**, **INT()**, **LEFT$()**, **MID$()** and **VAL()**.

API calls are functions. We give the API function a list of arguments and it returns a value, which we store in a variable. API functions are similar to Liberty BASIC's built-in functions, but the syntax is a little different. API functions are accessed with the **CALLDLL** statement.

CALLDLL

The parts of the CALLDLL statement are separated by commas. We'll discuss the parts of CALLDLL in order. The first part of CALLDLL refers to the DLL to be used. Windows contains many DLLs that can be accessed in Liberty BASIC programs.

The **CALLDLL** statement first needs to know which DLL is to be used. Many of the DLLs in Windows are recognized by Liberty BASIC without a need to open them. We refer to them by their handles. A very common and useful DLL is called "user32.DLL". Liberty BASIC gives us a handle for this DLL. It is "#user32". The first part of an API call to user32.dll looks like this:

```
CALLDLL #user32,
```

Take note that **CALLDLL** statements must be on a single line. We often split them into multiple lines to make them easier to read. We do this by using the underscore line continuation character as we did in the demonstration program above. It allows us to write a statement of code on multiple lines so that we can add comments and avoid the need to read long lines of code, but Liberty BASIC still sees the statement as a single line of code.

Here is an API call on one line of code:

```
CallDLL #user32, "IsWindowEnabled", hButton as uLong, result As Long
```

Here is the same API call broken into several lines with the underscore line continuation character. This allows us to add comments to each argument.

```
CallDLL #user32, "IsWindowEnabled",_
   hButton as uLong,_   'handle of window or control
   result As Long       'nonzero = enabled
```

API Function Names

The next part of the **CALLDLL** statement is the name of the API function. It is a text string and it must be enclosed in quotation marks. The name is case sensitive, so "FunctionName" is not the same as "FUNCTIONNAME". Be sure to copy the capitalization properly. A generic CALLDLL statement looks like this:

```
calldll #dll, "FunctionName",...
```

An actual API function called "IsWindowEnabled" looks like this:

```
CallDLL #user32, "IsWindowEnabled",...
```

API Arguments

Each API call has a set list of arguments separated by commas. Each of these arguments gives the API function some information. The function evaluates this information. It performs actions and it returns a value based on the information in these arguments. Arguments can be literal numbers or strings, or they can be numeric or string variables. They **cannot** be array elements or expressions.

Arguments are passed "as type". You can read more about types later in this tutorial.

Acceptable arguments that use literal numbers or strings, or numeric or string variables:

```
calldll #dll, "FunctionName", argument1 as type1,...
calldll #dll, "FunctionName", 23 as type2,...
calldll #dll, "FunctionName", "Some Text" as type3,...
var$ = "A bit of text."
calldll #dll, "FunctionName", var$ as type4,...
```

Some unacceptable arguments that include expressions and array elements:

```
calldll #dll, "FunctionName", array(1) as type1,...
calldll #dll, "FunctionName", 23 + 16 as type2,...
calldll #dll, "FunctionName", "Some Text" + "More Text" as type3,...
var$ = "A bit of text."
calldll #dll, "FunctionName", var$ + "hello" as type4,...
```

Some API functions require many arguments. Arguments are separated by commas. A function with three arguments looks like this:

```
calldll #dll, "FunctionName", argument1 as type1,
 argument2 as type2, argument3 as type3,...
```

In the demo for this tutorial, there is only one argument. The next tutorial will discuss API functions that require multiple arguments.

Handles

Many API calls require the **Windows handle** of a program window or control as one of the arguments.This Windows handle is a number of type ULONG. It is not the same as the "#handle" Liberty BASIC handle. We retrieve the Windows handle with the **HWND()** function and store it in a variable. In this example, the variable is called "hButton".

```
button #1.quit, "Quit", [quit], UL, 10, 60
open "API Demo" for window as #1

'get the handle of the quit button
hButton = HWND(#1.quit)
```

Handle to Graphics Window

Windows of type "graphics" are a special case. The handle returned by the HWND() function is the handle to the graphics area of the window. To retrieve the handle of the actual window, use the GetParent API call. You'll need this parent handle to communicate with the window itself to do things like move/resize, change caption, etc.

```
open "Demo" for graphics as #gr
handle = hwnd(#gr)

calldll #user32, "GetParent", _
handle as ulong, _  'get parent of this window
parent as ulong     'returns handle of parent window
```

Types

API functions need to know the **type** of each argument. Arguments can be numbers, strings or structs. We will talk about numeric arguments in this tutorial.

The most common numeric types are these. Types that hold numbers that are 4 bytes in size include ulong and long. Types that hold numbers that are 2 bytes in size include short, ushort, word and boolean. There are other types, but we won't discuss them in this tutorial.

A CALLDLL statement looks like this with actual types used in the arguments:

```
calldll #dll, "FunctionName", argument1 as ulong,
 argument2 as word, argument3 as long,...
```

Handle Types

In Liberty BASIC, Windows handles must always be passed as either "ulong" type or "word" type. In the demonstration program at the start of this tutorial, the "hButton" handle is passed "as ulong".

Return Values

API functions return a value, and we must include the "as type" for that value, just as we did for the list of arguments. The numeric return types include ulong, long, short, word, ushort and boolean.

Here is a generic call that includes the return value. The returned value is always the last part of the **CALLDLL** statement and it is separated from the list of arguments by a comma.

```
calldll #dll, "FunctionName", argument1 as ulong,
 argument2 as word, result as long
```

Note that you cannot use a variable name for the return type that is called "return" because that is a Liberty BASIC statement. Variable names cannot be the same as statement or function names in the Liberty BASIC language. Liberty BASIC programmers often use a variable name of "result" to contain the value returned by the API function.

Our programs might need to know the value returned by the function so that they can perform the proper actions. Use "if/then" to manage program flow after an API call has been made, as we've done in the demonstration program below.

Demo

Here again is the demonstration program. It opens a Liberty BASIC window that contains two buttons. It retrieves the handle of the Quit button with the **HWND()** function. It passes that handle into a user32.dll function called "IsWindowEnabled". The IsWindowEnabled function returns a value telling the program whether the window or control is enabled at the time the function is called.

The handle is passed as type "ulong". The return type is long. The value is either nonzero (meaning it is true) or zero (meaning it is false.) If IsWindowEnabled returns a value of 0, the window is not enabled. If it returns any other value than 0, the window is enabled when the function is called.

If the quit button is enabled, the program disables it. If it is disabled, the program enables it.

```
nomainwin

'open a program window
button #1.enable, "Enable/Disable",[doEnable],UL,10,10
button #1.quit, "Quit", [quit], UL, 10, 60
open "API Demo" for window as #1
#1 "trapclose [quit]"

'get the handle of the quit button
hButton = HWND(#1.quit)

wait

[doEnable]
```

```
'The IsWindowEnabled function determines whether the
'specified window or control is enabled
'for mouse and keyboard input

CallDLL #user32, "IsWindowEnabled",_
   hButton as uLong,_   'handle of window or control
   result As Long       'nonzero = enabled

'a result of 0 means that the window is NOT enabled
'a nonzero result means that the window IS enabled

if result = 0 then
    'if button was disabled, enable it
    #1.quit "!enable"
else
    'if bitton was enabled, disable it
    #1.quit "!disable"
end if

wait

[quit] close #1 :  end
```

The next tutorial in this series will demonstrate how to use multiple arguments in an API call to do something we can't do in Liberty BASIC.

Table of Contents

Handle to Graphics Window

Types

Handle Types

Return Values

Demo