Multiple-Timer Routines

May23, 2011
By -
nukesrus21 (Brandon Parker)

Table of Contents

Multiple-Timer Routines

Native Liberty Basic Timer Command

Timer Objectives

CallBacks and Associated Functions

Creating a Timer Queue

Deleting a Timer Queue and all Timer Queue Timers

Creating a Timer Queue Timer

Changing a Timer Queue Timer's Properties

Deleting Timer Queue Timers

Summary

Native Liberty Basic Timer Command

The native Liberty Basic Timer command manages a single Windows timer. While this is useful for controlling program flow at certain intervals there are a few drawbacks.

1. There is ONLY ONE Timer.
2. There are also two bugs associated with the Timer command that cause issues found here: Liberty Basic Bug Tracker - Events
3. Timer events will build up if the program is busy doing something else or if a Notice Dialog is issued and is not acnkowledged prior to the Timer firing.

Here is the example from the Liberty Basic Helpfile:

```
'set a timer to fire in 3 seconds
 'using branch label event handler
 timer 3000, [itHappened]
 'wait here
 wait

[itHappened]
 'deactivate the timer
 timer 0
 confirm "It happened! Do it again?"; answer
 if answer then
 'reactivate the timer
 timer 3000, [itHappened]
 wait
 end if
 end
```

As you can see the code to utilize the native Timer command is very simple and easy to use. Were it not for the drawbacks mentioned previously we would not require anything further. That leads me to the topic of this article.

Timer Objectives

We should start by stating the objectives for our timer.

1. The user should be able to create multiple timers.
2. The timer should work as well or better than the native Liberty Basic Timer.
3. The timer should be easy to use and have an adjustable interval.
4. The timer should have the capability of being removed during program execution.
5. The timer events should not build up, but should cause the program to branch off and return when completed.
6. The timer should be destroyed when the program execution has completed.

This is not exactly the simplest to do, but it can be acoomplished as we will soon see.

Introducing the Timer Queue Functions. These functions are Kernel32.dll which is part of the Windows API. These functions make it possible for a Liberty Basic program to take advantage of Windows' Thread Pooling capabilities where we can basically allow "worker threads" that are managed by Windows to control our timers and the CallBacks to functions within the program. Not only does this approach make it fairly simple to use multiple timers. It also prevents our Liberty Basic program from ever just sitting idle while we peruse menus, context menus, or drag the window around (this last statement occurs if you use a loop with a kernel sleep for timing not with the native Timer command).

Now that we have a general idea of what will be happening we should get right down to buisiness and start learning how to use a few wrapper functions for these API calls.

In order to use the functions that will follow we will need to set up a couple structures and some constants that we will be using along with a few other values that will be used for the function calls. They are as follows:

```
'The first Structure is for holding the handle
'to the created Timer Queue
Struct TimerQueue, handle As ulong

'The Second Structure is required for returning the
'handle of the timer
Struct phNewTimer, handle As ulong

'Now we will define a few flags that can be used.
WT.EXECUTEDEFAULT = _NULL
'By default, the callback function is queued to a non-
I/O worker thread.

WT.EXECUTEONLYONCE = 8
'The timer will be set to the signaled state only once.
 'If this flag is set, the Period parameter must be zero.

DueTime = 500
Period = 1500
Flags = WT.EXECUTEDEFAULT OR WT.EXECUTELONGFUNCTION
```

Now that we have that all set. We should probably create a windows with a few things in it to play with. Note that all of Dynamic Array Functions have been included to simplify the expandability of the functions.

```
NoMainWin
Global True : True = 1
Global False : False = 0
WindowWidth = 270
WindowHeight = 150
StaticText #Test.timer1, "", 90, 25, 100, 25
StaticText #Test.timer2, "", 90, 55, 100, 25
StaticText #Test.timer3, "", 90, 85, 100, 25
Open "Multiple Timers via API" For Window As #Test
Print #Test, "TrapClose Quit"
```

Ok; so we have our window all set up. Should we get started on creating timers? Well, not just yet. We should go right ahead and set up the CallBacks and associated Functions that our timers will be utilizing that way we are all set to go once we create the timers.

CallBacks and Associated Functions

At this point if you are not familiar with CallBacks it would probably be a good idea to take a step back and study them prior to returning to this tutorial. For the sake of simplicity and to shorten this I will just provide the CallBacks and associated Functions rather than discussing the how/ why of them. Although I would like to mention that these CallBacks require only two arguments; sending them more will cause failure and their return types are Void although this is not necessarily set in stone.

```
CallBack WaitorTimerCallBackAdd, WaitorTimerCallBack(ULong,ULong),Void
CallBack WaitorTimerCallBack2Add,
 WaitorTimerCallBack2(ULong,ULong),Void
CallBack WaitorTimerCallBack3Add,
 WaitorTimerCallBack3(ULong,ULong),Void

Function WaitorTimerCallBack(ULong,ULong)
 Print #Test.timer3, ""
 Print #Test.timer1, "Timer #1 Fired"
End Function

Function WaitorTimerCallBack2(ULong,ULong)
 Print #Test.timer1, ""
 Print #Test.timer2, "Timer #2 Fired"
End Function

Function WaitorTimerCallBack3(ULong,ULong)
 Print #Test.timer2, ""
 Print #Test.timer3, "Timer #3 Fired"
End Function
```

The CallBacks should be placed at the beginning of the program prior to attempting to create any Timer Queue Timers.

Creating a Timer Queue

We will start by creating a timer queue for our program where all of our timers will reside. Although this is not necessary if you will only be creating a few timers it never hurts for a program to have its own timer queue. In order to do this we will need to call the CreateTimerQueue Function from the Kernel32.dll.

Here is the wrapper function we will be using:

```
Function CreateTimerQueue()
 CallDLL #kernel32, "CreateTimerQueue", CreateTimerQueue As ulong
End Function
```

Notice that the CreateTimerQueue() Function we have created requires no parameters, but does return a value; the handle to the Timer Queue. This value we will need to keep so we will store it in our TimerQueue Structure. We will call the CreateTimerQueue() Function in the following manner which will allow us to store the beforementioned handle to the Timer Queue.

```
TimerQueue.handle.struct = CreateTimerQueue()
```

And that is all you have to do to create a Timer Queue for a Liberty Basic program. With that being said before we jump into creating Timer Queue Timers we first should look at how to destroy the Timer Queue and any Timer Queue Timers because if we attempt to close our program without doing so we will have a huge mess on our hands as the thread(s) in the thread pool that are controlling our Timer(s) will not know that the program has been closed and will attempt to activate the code for the Timer(s). This normally will cause a few pop-ups from Windows and will cause the program to crash (even the Liberty Basic IDE) all the way to the Desktop.

Deleting a Timer Queue and all Timer Queue Timers

We will do destroy the Timer Queue and all Timer Queue Timers by calling the DeleteTimerQueueEx Function from the Kernel32.dll.

Here is the wrapper function we will be using:

```
Function DeleteTimerQueueEx(TimerQueue, CompletionEvent)
 CallDLL #kernel32, "DeleteTimerQueueEx", TimerQueue As ulong, _
 CompletionEvent As ulong, _
 DeleteTimerQueueEx As ulong
 result$ = ReDimOrSetStringArray$("TimerQueueTimer", "NULL*", 0)
```

```
End Function
```

The DeleteTimerQueueEx() Function takes two arguments as input; TimerQueue and CompletionEvent. TimerQueue is the handle to the Timer Queue. Remember from earlier that we stored that handle in TimerQueue.handle.struct when we called the CreateTimerQueue() Function. CompletionEvent is basically what the name describes; a handle to a completion event that will be called when all Timer Event have completed. We will be passing _INVALID_HANDLE_VALUE as the argument for CompletionEvent as this causes the function to wait for all callback functions to complete before returning. The function returns a non-zero value if successful and zero otherwise. This will ensure that all of our expected timed events are taken care of and makes for a very clean closing process for the Timer Queue altogether.

To illustrate how this would be performed when closing a window here is a subroutine that would be used to close our program. Notice that we make a call to the DeleteTimerQueueEx() Function prior to closing our window.

```
Sub Quit handle$
 result = DeleteTimerQueueEx(TimerQueue.handle.struct,
 _INVALID_HANDLE_VALUE)
 Close #handle$
 End
End Sub
```

Creating a Timer Queue Timer

Now we can get to the good part.... Creating Timers!!

We will accomplish this by calling the CreateTimerQueueTimer Function from the Kernel32.dll.

As we examine the wrapper function CreateTimerQueueTimer() we will notice that the function expects six arguments.

1. TimerName$ - This can be any name you wish to give the timer and is used with the Dynamic Array Functions.
2. TimerQueue - This is the handle to the Timer Queue that we created by calling the CreateTimerQueue() Function.
3. WaitorTimerCallBackAdd - This is the address of the CallBack Function we want the Timer to call.
4. DueTime - This is the amount of time after the Timer is created that the process waits prior to calling the supplied Function in milliseconds.
5. Period - This is the Timer Interval.
6. Flags - This OR'd value is a combination of flags that tells Windows how the Timer should work.

(You will find a list of some, but not all possible values in the full example below.)

The function returns a non-zero value if successful and zero otherwise. The function will also return the handle to the newly created Timer in phNewTimer.handle.struct if it is successful.

You can also see that the Dynamic Array Function within the CreatetimerQueueTimer() Function will be storing the Name, Handle, and Period of any Timer that we create inside the TimerQueueTimer$() Array.

```
Function CreateTimerQueueTimer(TimerName$, TimerQueue,
 WaitorTimerCallBackAdd, DueTime, Period, Flags)
 CallDLL #kernel32, "CreateTimerQueueTimer", phNewTimer As struct, _
 TimerQueue As ulong, _
 WaitorTimerCallBackAdd As ulong, _
 parameter As long, _
 DueTime As ulong, _
 Period As ulong, _
 Flags As ulong, _
 CreateTimerQueueTimer As long
 TimerQueueTimer$(
NextAvailableElementStringArray("TimerQueueTimer", False)) =
 TimerName$ + " " + _
 str$(phNewTimer.handle.struct) + " " + _
 str$(Period)
End Function
```

Ok, now that we have the function we should create a few timers to get things off to a good start. We will create three timers for the three CallBacks and associated Functions that we set up in the beginning of this tutorial.

```
result = CreateTimerQueueTimer("Timer1", TimerQueue.handle.struct, _
 WaitorTimerCallBackAdd, DueTime, Period, Flags)
result = CreateTimerQueueTimer("Timer2", TimerQueue.handle.struct, _
 WaitorTimerCallBack2Add, _
 DueTime + Int(Period * .3), Period, Flags)
result = CreateTimerQueueTimer("Timer3", TimerQueue.handle.struct, _
 WaitorTimerCallBack3Add, _
 DueTime + Int(Period * .6), Period, Flags)
```

Now that we have created a few timers we should see how to change a Timer. That can be accomplished by calling the ChangeTimerQueueTimer Function from the Kernel32.dll.

Changing a Timer Queue Timer's Properties

Here we have the wrapper function:
Warning: This function should not be used and should be replaced by the ChangeTimerQueueTimerbyName() Function if the Dynamic Array Functions are in use as it will not update the TimerQueueTimer$() Array.

```
Function ChangeTimerQueueTimer(TimerQueue, hTimer, DueTime, Period)
 CallDLL #kernel32, "ChangeTimerQueueTimer", TimerQueue As ulong, _
 hTimer As ulong, _
 DueTime As ulong, _
 Period As ulong, _
 ChangeTimerQueueTimer As ulong
End Function
```

Examining the ChangeTimerQueueTimer() Function we find that it requires four inputs:

1. TimerQueue - This is the handle to the Timer Queue that we created by calling the CreateTimerQueue() Function.
2. hTimer - This is the handle to the Timer that we want to change.
3. DueTime - This is the amount of time after the Timer is changed that the process waits prior to calling the supplied Function in milliseconds.
4. Period - This is the NEW Timer Interval.

The function returns a non-zero value if successful and zero otherwise.

This is an example of how to call the function:

```
result = ChangeTimerQueueTimer(TimerQueue.handle.struct, hTimer,
 DueTime, Period)
```

Now keeping track of all of those Timer handles would be a pain had we not set everything up to where the Dynamic Array Functions would storing our Timer information inside the TimerQueueTimer$() Array. This makes it possible to create another function that calls the ChangeTimerQueueTimer() Function where we supply the Name of the Timer instead of the handle.

Here is that function; ChangeTimerQueueTimerbyName():

```
Function ChangeTimerQueueTimerbyName(TimerName$, TimerQueue,
 DueTime, Period)
 For i = 0 To UBoundStringArray("TimerQueueTimer")
 If (TimerQueueTimer$(i) = "") Then Exit For
 If TimerName$ = Word$(TimerQueueTimer$(i), 1) Then
 TimerQueueTimer$(i) = Word$(TimerQueueTimer$(i), 1) + " " + _
 Word$(TimerQueueTimer$(i), 2) + " " + _
 str$(Period)
 ChangeTimerQueueTimerbyName = ChangeTimerQueueTimer(TimerQueue, _
 Val(Word$(TimerQueueTimer$(i), 2)), _
 DueTime, Period)
 Exit For
 End If
 Next i
End Function
```

This is how you would change a Timer by Name:

```
result = ChangeTimerQueueTimerbyName("Timer1",
 TimerQueue.handle.struct, DueTime, Period)
```

Deleting Timer Queue Timers

With respect to deleting timers there have been two functions created pretty much the same as with changing the timers. The first wrapper function will delete a Timer based on its handle and the second function will delete a Timer based on the name and is dependent upon the first function, but provides extra flexibility.

The first is the DeleteTimerQueueTimer() Function:

```
Function DeleteTimerQueueTimer(TimerQueue, hTimer, CompletionEvent)
 CallDLL #kernel32, "DeleteTimerQueueTimer", TimerQueue As ulong, _
 hTimer As ulong, _
 CompletionEvent As ulong, _
 DeleteTimerQueueTimer As long
End Function
```

The second is the DeleteTimerQueueTimerbyName() Function:

```
'CompletionEvent can be _NULL or _INVALID_HANDLE_VALUE (recommend the
latter)
Function DeleteTimerQueueTimerbyName(timerName$, TimerQueue,
 CompletionEvent)
 For i = 0 To UBoundStringArray("TimerQueueTimer")
 If (TimerQueueTimer$(i) = "") Then Exit For
 If Word$(TimerQueueTimer$(i), 1) = timerName$ Then
 DeleteTimerQueueTimerbyName = DeleteTimerQueueTimer(TimerQueue, _
 Val(Word$(TimerQueueTimer$(i), 2)), _
 CompletionEvent)
 TimerQueueTimer$(i) = ""
 result = CompactStringArray("TimerQueueTimer")
 Exit For
 End If
 Next i
End Function
```

Both functions should be called as follows:

1. DeleteTimerQueueTimer() Function

```
result = DeleteTimerQueueTimer(TimerQueue.handle.struct, hTimer,
 CompletionEvent
```

2. DeleteTimerQueueTimerbyName() Function

```
result = DeleteTimerQueueTimerbyName("Timer1",
 TimerQueue.handle.struct, CompletionEvent)
```

With either one of these calls we will be passing _INVALID_HANDLE_VALUE as the argument for CompletionEvent as this causes the function to wait for all callback functions to complete before returning. These functions return a non-zero value if successful and zero otherwise.

The last function to discuss is the TimerInterval() Function. This function can only be used if the Dynamic Array Functions are being used. It requires a single parameter as its input; the Timer Name that the user gave to the timer when it was created. The function returns the Timer Inverval associated with that Timer in milliseconds.

```
Function TimerInterval(timerName$)
 For i = 0 To UBoundStringArray("TimerQueueTimer")
 If (TimerQueueTimer$(i) = "") Then Exit For
 If Word$(TimerQueueTimer$(i), 1) = timerName$ Then
 TimerInterval = Val(Word$(TimerQueueTimer$(i), 3))
 Exit For
 End If
 Next i
End Function
```

Please note that the Dynamic Array Functions that are used in the functions above can be found through the following link:
Dynamic Arrays

Summary

Pheeewww!!! That's about all it takes to make multiple timers. It seems like a lot of work, but trust me once you understand how everything fits together you'll be creating timers left and right in no time. Below you will find a working demo which uses some of the functions to create three timers that change what is being displayed in the window. I hope this has been at least a little helpful...

{:0)

Brandon Parker

Timer Queue Functions (Multiple Timers).bas

- Details
- Download
- 31 KB

Multiple-Timer Routines | Native Liberty Basic Timer Command | Timer Objectives | CallBacks and Associated Functions | Creating a Timer Queue | Deleting a Timer Queue and all Timer Queue Timers | Creating a Timer Queue Timer | Changing a Timer Queue Timer's Properties | Deleting Timer Queue Timers | Summary