Thread: [VB6] - Module for working with multithreading.

Jun 12th, 2018, 03:05 PM #1

The trick

Thread Starter
Frenzied Member

Join Date: Feb 2015
Posts: 1,045

[VB6] - Module for working with multithreading.

Hello everyone!

I present the module for working with multithreading in VB6 for Standard EXE projects. This module is based on this solution with some bugfixing and the new functionality is added. The module doesn't require any additional dependencies and type libraries, works as in the IDE (all the functions work in the main thread) as in the compiled form.

To start working with the module, you need to call the **Initialize** function, which initializes the necessary data (it initializes the critical sections for exclusive access to the heaps of marshalinig and threads, modifies **VBHeader** (here is description), allocates a **TLS** slot for passing the parameters to the thread).

The main function of thread creation is **vbCreateThread**, which is an analog of the CreateThread function.

Code:

```
' // Create a new thread
Public Function vbCreateThread(ByVal lpThreadAttributes As Long, _
                               ByVal dwStackSize As Long,  _
                               ByVal lpStartAddress As Long, _
                               ByVal lpParameter As Long,  _
                               ByVal dwCreationFlags As Long, _
                               ByRef lpThreadId As Long, _
                               Optional ByVal bIDEInSameThread As Boolean = True) As Long
```

The function creates a thread and calls the function passed in the **lpStartAddress** parameter with the **lpParameter** parameter.
In the IDE, the call is reduced to a simple call by the pointer implemented through DispCallFunc. In the compiled form, this function works differently. Because a thread requires initialization of project-specific data and initialization of the runtime, the parameters passed to **lpStartAddress** and **lpParameter** are temporarily stored into the heap by the **PrepareData** function, and the thread is created in the **ThreadProc** function, which immediately deals with the initialization and calling of the user-defined function with the user parameter. This function creates a copy of the **VBHeader** structure via **CreateVBHeaderCopy** and changes the public variable placement data in the **VbPublicObjectDescriptor.lpPublicBytes**, **VbPublicObjectDescriptor.lpStaticBytes** structures (BTW it wasn't implemented in the previous version) so that global variables are not affected during initialization. Further, **VBDllGetClassObject** calls the **FakeMain** function (whose address is written to the modified **VBHeader** structure). To transfer user parameters, it uses a **TLS** slot (since **Main** function doesn't accept parameters, details here). In **FakeMain**, parameters are directly extracted from **TLS** and a user procedure is called. The return value of the function is also passed back through **TLS**. There is one interesting point related to the copy of the header that wasn't included in the previous version. Because the runtime uses the header after the thread ends (with **DLL_THREAD_DETACH**), we can't release the header in the **ThreadProc** procedure, therefore there will be a memory leak. To prevent the memory leaks, the heap of fixed size is used, the headers aren't cleared until there is a free memory in this heap. As soon as the memory ends (and it's allocated in the **CreateVBHeaderCopy** function), resources are cleared. The first **DWORD** of header actually stores the **ID** of the thread which it was created in and the **FreeUnusedHeaders** function checks all the headers in the heap. If a thread is completed, the memory is freed (although the **ID** can be repeated, but this doesn't play a special role, since in any case there will be a free memory in the heap and if the header isn't freed in one case, it will be released later). Due to the fact that the cleanup process can be run immediately from several threads, access to the cleanup is shared by the critical section **tLockHeap.tWinApiSection** and if some thread is already cleaning up the memory the function will return True which means that the calling thread should little bit waits and the memory will be available.

The another feature of the module is the ability to initialize the runtime and the project and call the callback function. This can be useful for callback functions that can be called in the context of an arbitrary thread (for example, **InternetStatusCallback**). To do this, use the **InitCurrentThreadAndCallFunction** and **InitCurrentThreadAndCallFunctionIDEProc** functions. The first one is used in the compiled application and takes the address of the callback function that will be called after the runtime initialization, as well as the parameter to be passed to this function. The address of the first parameter is passed to the callback procedure to refer to it in the user procedure:

Code:

```
' // This function is used in compiled form
Public Function CallbackProc(  _
                ByVal lThreadId As Long, _
                ByVal sKey As String,  _
                ByVal fTimeFromLastTick As Single) As Long
    ' // Init runtime and call CallBackProc_user with VarPtr(lThreadId) parameter
    InitCurrentThreadAndCallFunction AddressOf CallBackProc_user, VarPtr(lThreadId), Callk
End Function

' // Callback function is called by runtime/window proc (in IDE)
Public Function CallBackProc_user(  _
                ByRef tParam As tCallbackParams) As Long
```

```
End Function
```

CallBackProc_user will be called with the initialized runtime.

This function doesn't work in the **IDE** because in the **IDE** everything works in the main thread. For debugging in the **IDE** the function **InitCurrentThreadAndCallFunctionIDEProc** is used which returns the address of the assembler thunk that translates the call to the main thread and calls the user function in the context of the main thread. This function takes the address of the user's callback function and the size of the parameters in bytes. It always passes the address of the first parameter as a parameter of a user-defined function. I'll tell you a little more about the work of this approach in the **IDE**. To translate a call from the calling thread to the main thread it uses a message-only window. This window is created by calling the **InitializeMessageWindow** function. The first call creates a **WindowProc** procedure with the following code:

Code:

```
    CMP DWORD [ESP+8], WM_ONCALLBACK
    JE SHORT L
    JMP DefWindowProcW
L:  PUSH DWORD PTR SS:[ESP+10]
    CALL DWORD PTR SS:[ESP+10]
    RETN 10
```

As you can see from the code, this procedure "listens" to the **WM_ONCALLBACK** message which contains the parameter **wParam** - the function address, and in the **lParam** parameters. Upon receiving this message it calls this procedure with this parameter, the remaining messages are ignored. This message is sent just by the assembler thunk from the caller thread. Futher, a window is created and the handle of this window and the code heap are stored into the data of the window class. This is used to avoid a memory leak in the **IDE** because if the window class is registered once, then these parameters can be obtained in any debugging session. The callback function is generated in **InitCurrentThreadAndCallFunctionIDEProc**, but first it's checked whether the same callback procedure has already been created (in order to don't create the same thunk). The thunk has the following code:

Code:

```
LEA EAX, [ESP+4]
PUSH EAX
PUSH pfnCallback
PUSH WM_ONCALLBACK
PUSH hMsgWindow
Call SendMessageW
RETN lParametersSize
```

As you can see from the code, during calling a callback function, the call is transmitted via **SendMessage** to the main thread. The **lParametersSize** parameter is used to correctly restore the stack.

The next feature of the module is the creation of objects in a separate thread, and you can create them as private objects (the method is based on the code of the NameBasedObjectFactory by firehacker module) as public ones. To create the project classes use the **CreatePrivateObjectByNameInNewThread** function and for **ActiveX**-public classes **CreateActiveXObjectInNewThread** and **CreateActiveXObjectInNewThread2** ones. Before creating instances of the project classes you must first enable marshaling of these objects by calling the **EnablePrivateMarshaling** function. These functions accept the class identifier (**ProgID** / **CLSID** for **ActiveX** and the name for the project classes) and the interface identifier (**IDispatch** / **Object** is used by default). If the function is successfully called a marshaled object and an asynchronous call **ID** are returned. For the compiled version this is the **ID** of thread for **IDE** it's a pointer to the object. Objects are created and "live" in the **ActiveXThreadProc** function. The life of objects is controlled through the reference count (when it is equal to 1 it means only **ActiveXThreadProc** refers to the object and you can delete it and terminate the thread).
You can call the methods either synchronously - just call the method as usual or asynchronously - using the **AsynchDispMethodCall** procedure. This procedure takes an asynchronous call **ID**, a method name, a call type, an object that receives the call notification, a notification method name and the list of parameters. The procedure copies the parameters to the temporary memory, marshals the notification object, and sends the data to the object's thread via **WM_ASYNCH_CALL**. It should be noted that marshaling of parameters isn't supported right now therefore it's necessary to transfer links to objects with care. If you want to marshal an object reference you should use a synchronous method to marshal the objects and then call the asynchronous method. The procedure is returned immediately. In the **ActiveXThreadProc** thread the data is retrieved and a synchronous call is made via **MakeAsynchCall**. Everything is simple, **CallByName** is called for the thread object and **CallByName** for notification. The notification method has the following prototype:

Code:

```
Public Sub CallBack (ByVal vRet As Variant)
```

, where **vRet** accepts the return value of the method.

The following functions are intended for marshaling: **Marshal**, **Marshal2**, **UnMarshal**, **FreeMarshalData**. The first one creates information about the marshaling (**Proxy**) of the interface and puts it into the stream (**IStream**) that is returned. It accepts the interface identifier in the **pInterface** parameter (**IDispatch** / **Object** by default). The **UnMarshal** function, on the contrary, receives a stream and creates a **Proxy** object based on the information in the stream. Optionally, you can release the thread object. **Marshal2** does the same thing as **Marshal** except that it allows you to create a **Proxy** object many times in different threads. **FreeMarshalData** releases the data and the stream accordingly.
If, for example, you want to transfer a reference to an object between two threads, it is enough to call the **Marshal** / **UnMarshal** pair in the thread which created the object and in the thread that receives the link respectively. In another case, if for example there is the one global object and you need to pass a reference to it to the multiple threads (for example, the logging object), then **Marshal2** is called in the object thread, and **UnMarshal** with the **bReleaseStream** parameter is set to **False** is called in client threads. When the data is no longer needed, **FreeMarshalData** is called.

The **WaitForObjectThreadCompletion** function is designed to wait for the completion of the object thread and receives the **ID** of the asynchronous call. It is desirable to call this function always at the end of the main process because an object thread can somehow interact with the main thread and its objects (for example, if the object thread has a marshal link to the interface of the main thread).

The **SuspendResume** function is designed to suspend/resume the object's thread; **bSuspend** determines whether to sleep or resume the thread.

In addition, there are also several examples in the attacment of working with module:

1. **Callback** - the project demonstrates the work with the **callback**-function periodically called in the different threads. Also, there is an additional project of native dll (on VB6) which calls the function periodically in the different threads;
2. **JuliaSet** - the **Julia** fractal generation in the several threads (user-defined);
3. **CopyProgress** - Copy the folder in a separate thread with the progress of the copy;
4. **PublicMarshaling** - Creating public objects (**Dictionary**) in the different threads and calling their methods (synchronously / asynchronously);
5. **PrivateMarshaling** - Creating private objects in different threads and calling their methods (synchronously / asynchronously);
6. **MarshalUserInterface** - Creating private objects in different threads and calling their methods (synchronously / asynchronously) based on user interfaces (contains tlb and Reg-Free manifest).

The module is poorly tested so bugs are possible. I would be very glad to any bug-reports, wherever possible I will correct them.
Thank you all for attention!

Best Regards,
The trick.

Attached Files
VBMultithreadingModule.zip (47.2 KB, 50 views)

Last edited by The trick; Yesterday at 06:19 AM.

EXE loader|Inline assembler Add-in|Kernel-mode driver on VB6|Multithreading in StandartEXE|Native VB6-DLL
Code injection to other process|DLL injection|DirectX9|DirectSound|TrickControls|MP3 player from memory
Easy calling by pointer|Hash-table|Safe subclassing|Vocoder|Creation of native DLL|COM-DLL without registration
FM-synthesizer|FFT spectrum-analyser|3D Fir-tree|Asynch waiter|Wave steganography|Fast AVI-trimmer
Windows with custom rendering|String to integer and vice versa|Multithreading with marshaling|Owner-draw listbox mod
Advanced math|Library info|Create GIF-animation|Store data to self-EXE|Computer creates a music|TrickSound class

(rate this post) Reply Reply With Quote

Yesterday, 02:34 AM #2

xxdoc123
Addicted Member
Join Date: Aug 2016
Posts: 240

Re: [VB6] - Module for working with multithreading.

Originally Posted by **The trick**

Hello everyone!

I present the module for working with multithreading on VB6 for Standard EXE projects. This module is based on this solution with some bugfixing and the new functionality is added. The module doesn't require any additional dependencies and type libraries, works as in the IDE (all the functions work in the main thread) as in the compiled form.

*To start working with the module, you need to call the **Initialize** function, which initializes the necessary data (it initializes the critical sections for exclusive access to the heaps of marshalinig and threads, modifies **VBHeader** (here is description), allocates a **TLS** slot for passing the parameters to the thread).*

*The main function of thread creation is **vbCreateThread**, which is an analog of the CreateThread function.*

Code:

```
' // Create a new thread
Public Function vbCreateThread(ByVal lpThreadAttributes As Long, _
                               ByVal dwStackSize As Long, _
                               ByVal lpStartAddress As Long, _
                               ByVal lpParameter As Long, _
                               ByVal dwCreationFlags As Long, _
                               ByRef lpThreadId As Long, _
                               Optional ByVal bIDEInSameThread As Boolean = True) As Long
```

*The function creates a thread and calls the function passed in the **lpStartAddress** parameter with the **lpParameter** parameter.*
*In the IDE, the call is reduced to a simple call by the pointer implemented through DispCallFunc. In the compiled form, this function works differently. Because a thread requires initialization of project-specific data and initialization of the runtime, the parameters passed to **lpStartAddress** and **lpParameter** are temporarily stored into the heap by the **PrepareData** function, and the thread is created in the **ThreadProc** function, which immediately deals with the initialization and calling of the user-defined function with the user parameter. This function creates a copy of the **VBHeader** structure via **CreateVBHeaderCopy** and changes the public variable placement data in the **VbPublicObjectDescriptor.lpPublicBytes**, **VbPublicObjectDescriptor.lpStaticBytes** structures (BTW it wasn't implemented in the previous version) so that global variables are not affected during initialization. Further, **VBDllGetClassObject** calls the **FakeMain** function (whose address is written to the modified **VBHeader** structure). To transfer user parameters, it uses a **TLS** slot (since **Main** function doesn't accept parameters, details here). In **FakeMain**, parameters are directly extracted from **TLS** and a user procedure is called. The return value of the function is also passed back through **TLS**. There is one interesting point related to the copy of the header that wasn't included in the previous version. Because the runtime uses the header after the thread ends (with **DLL_THREAD_DETACH**), we can't release the header in the **ThreadProc** procedure, therefore there will be a memory leak. To prevent the memory leaks, the heap of fixed size is used, the headers aren't cleared until there is a free memory in this heap. As soon as the memory ends (and it's allocated in the **CreateVBHeaderCopy** function), resources are cleared. The first **DWORD** of header actually stores the **ID** of the thread which it was created in and the **FreeUnusedHeaders** function checks all the headers in the heap. If a thread is completed, the memory is freed (although the **ID** can be repeated, but this doesn't play a special role, since in any case there will be a free memory in the heap and if the header isn't freed in one case, it will be released later). Due to the fact that the cleanup process can be run immediately from several threads, access to the cleanup is shared by the critical section **tLockHeap.tWinApiSection** and if some thread is already cleaning up the memory the function will return True which means that the calling thread should little bit waits and the memory will be available.*

*The another feature of the module is the ability to initialize the runtime and the project and call the callback function. This can be useful for callback functions that can be called in the context of an arbitrary thread (for example, **InternetStatusCallback**). To do this, use the **InitCurrentThreadAndCallFunction** and **InitCurrentThreadAndCallFunctionIDEProc** functions. The first one is used in the compiled application and takes the address of the callback function that will be called after the runtime initialization, as well as the parameter to be passed to this function. The address of the first parameter is passed to the callback procedure to refer to it in the user procedure:*

Code:

```
' // This function is used in compiled form
Public Function CallbackProc(
                ByVal lThreadId As Long, _
                ByVal sKey As String, _
                ByVal fTimeFromLastTick As Single) As Long
    ' // Init runtime and call CallBackProc_user with VarPtr(lThreadId) parameter
    InitCurrentThreadAndCallFunction AddressOf CallBackProc_user, VarPtr(lThreadId), Calll
End Function

' // Callback function is called by runtime/window proc (in IDE)
Public Function CallBackProc_user(
                ByRef tParam As tCallbackParams) As Long

End Function
```

***CallBackProc_user** will be called with the initialized runtime.*

*This function doesn't work in the **IDE** because in the **IDE** everything works in the main thread. For debugging in the **IDE** the function **InitCurrentThreadAndCallFunctionIDEProc** is used which returns the address of the assembler thunk that translates the call to the main thread and calls the user function in the context of the main thread. This function takes the address of the user's callback function and the size of the parameters in bytes. It always passes the address of the first parameter as a parameter of a user-defined function. I'll tell you a little more about the work of this approach in the **IDE**. To translate a call from the calling thread to the main thread it uses a message-only window. This window is created by calling the **InitializeMessageWindow** function. The first call creates a **WindowProc** procedure with the following code:*

Code:

```
    CMP DWORD [ESP+8], WM_ONCALLBACK
    JE SHORT L
    JMP DefWindowProcW
L:  PUSH DWORD PTR SS:[ESP+10]
    CALL DWORD PTR SS:[ESP+10]
    RETN 10
```

*As you can see from the code, this procedure "listens" to the **WM_ONCALLBACK** message which contains the parameter **wParam** - the function address, and in the **lParam** parameters. Upon receiving this message it calls this procedure with this parameter, the remaining messages are ignored. This message is sent just by the assembler thunk*

*from the caller thread. Futher, a window is created and the handle of this window and the code heap are stored into the data of the window class. This is used to avoid a memory leak in the **IDE** because if the window class is registered once, then these parameters can be obtained in any debugging session. The callback function is generated in **InitCurrentThreadAndCallFunctionIDEProc**, but first it's checked whether the same callback procedure has already been created (in order to don't create the same thunk). The thunk has the following code:*

Code:

```
LEA EAX, [ESP+4]
PUSH EAX
PUSH pfnCallback
PUSH WM_ONCALLBACK
PUSH hMsgWindow
Call SendMessageW
RETN lParametersSize
```

*As you can see from the code, during calling a callback function, the call is transmitted via **SendMessage** to the main thread. The **lParametersSize** parameter is used to correctly restore the stack.*

*The next feature of the module is the creation of objects in a separate thread, and you can create them as private objects (the method is based on the code of the NameBasedObjectFactory by firehacker module) as public ones. To create the project classes use the **CreatePrivateObjectByNameInNewThread** function and for **ActiveX**-public classes **CreateActiveXObjectInNewThread** and **CreateActiveXObjectInNewThread2** ones. Before creating instances of the project classes you must first enable marshaling of these objects by calling the **EnablePrivateMarshaling** function. These functions accept the class identifier (**ProgID** / **CLSID** for **ActiveX** and the name for the project classes) and the interface identifier (**IDispatch** / **Object** is used by default). If the function is successfully called a marshaled object and an asynchronous call **ID** are returned. For the compiled version this is the **ID** of thread for **IDE** it's a pointer to the object. Objects are created and "live" in the **ActiveXThreadProc** function. The life of objects is controlled through the reference count (when it is equal to 1 it means only **ActiveXThreadProc** refers to the object and you can delete it and terminate the thread).*
*You can call the methods either synchronously - just call the method as usual or asynchronously - using the **AsynchDispMethodCall** procedure. This procedure takes an asynchronous call **ID**, a method name, a call type, an object that receives the call notification, a notification method name and the list of parameters. The procedure copies the parameters to the temporary memory, marshals the notification object, and sends the data to the object's thread via **WM_ASYNCH_CALL**. It should be noted that marshaling of parameters isn't supported right now therefore it's necessary to transfer links to objects with care. If you want to marshal an object reference you should use a synchronous method to marshal the objects and then call the asynchronous method. The procedure is returned immediately. In the **ActiveXThreadProc** thread the data is retrieved and a synchronous call is made via **MakeAsynchCall**. Everything is simple, **CallByName** is called for the thread object and **CallByName** for notification. The notification method has the following prototype:*

Code:

```
Public Sub CallBack (ByVal vRet As Variant)
```

*, where **vRet** accepts the return value of the method.*

*The following functions are intended for marshaling: **Marshal**, **Marshal2**, **UnMarshal**, **FreeMarshalData**. The first one creates information about the marshaling (**Proxy**) of the interface and puts it into the stream (**IStream**) that is returned. It accepts the interface identifier in the **pInterface** parameter (**IDispatch** / **Object** by default). The **UnMarshal** function, on the contrary, receives a stream and creates a **Proxy** object based on the information in the stream. Optionally, you can release the thread object. **Marshal2** does the same thing as **Marshal** except that it allows you to create a **Proxy** object many times in different threads. **FreeMarshalData** releases the data and the stream accordingly.*
*If, for example, you want to transfer a reference to an object between two threads, it is enough to call the **Marshal** / **UnMarshal** pair in the thread which created the object and in the thread that receives the link respectively. In another case, if for example there is the one global object and you need to pass a reference to it to the multiple threads (for example, the logging object), then **Marshal2** is called in the object thread, and **UnMarshal** with the **bReleaseStream** parameter is set to **False** is called in client threads. When the data is no longer needed, **FreeMarshalData** is called.*

*The **WaitForObjectThreadCompletion** function is designed to wait for the completion of the object thread and receives the **ID** of the asynchronous call. It is desirable to call this function always at the end of the main process because an object thread can somehow interact with the main thread and its objects (for example, if the object thread has a marshal link to the interface of the main thread).*

*The **SuspendResume** function is designed to suspend/resume the object's thread; **bSuspend** determines whether to sleep or resume the thread.*

In addition, there are also several examples in the attacment of working with module:

1. ***Callback** - the project demonstrates the work with the **callback**-function periodically called in the different threads. Also, there is an additional project of native dll (on VB6) which calls the function periodically in the different threads;*
2. ***JuliaSet** - the **Julia** fractal generation in the several threads (user-defined);*
3. ***CopyProgress** - Copy the folder in a separate thread with the progress of the copy;*
4. ***PublicMarshaling** - Creating public objects (**Dictionary**) in the different threads and calling their methods (synchronously / asynchronously);*
5. ***PrivateMarshaling** - Creating private objects in different threads and calling their methods (synchronously / asynchronously);*
6. ***MarshalUserInterface** - Creating private objects in different threads and calling their methods (synchronously / asynchronously) based on user interfaces (contains tlb and Reg-Free manifest).*

The module is poorly tested so bugs are possible. I would be very glad to any bug-reports, wherever possible I will correct them.
Thank you all for attention!

Best Regards,
The trick.

Really a very useful example

(rate this post) **Reply** **Reply With Quote**

Yesterday, 08:06 AM #3

DEXWERX

Frenzied Member

Join Date: Jun 2015
Posts: 1,954

Re: [VB6] - Module for working with multithreading.

Now this is a lot of work.... It basically sums up all the correct/stable methods needed, and includes the most advanced methods available (developed by Curland, Trick, and firehacker) as far as threading in VB.
well done.

edit: this is top. I definitely appreciate the work that went in to creating a stable asynchronous framework (in any language). Thanks for working out all the more advance methods, and then putting it into something usable for most anyone. Just a great contribution. Thanks Trick.

edit 2: also I was going to ask you to work out using a typelib embedded in a resource for marshaling a custom interface...
reg-free. but you already included it (mostly). ridiculous.

Last edited by DEXWERX; Yesterday at 09:12 AM.

Imagine what it would be like to set breakpoints in, or step through subclassing code;
and then being able to hit stop/end/debug or continue, without crashing the IDE.
VB6.tlb | Bulletproof Subclassing in the IDE (no thunks/assembly/DEP issues)

(rate this post) **Reply** **Reply With Quote**

Yesterday, 09:12 AM #4

DEXWERX
Frenzied Member
Join Date: Jun 2015
Posts: 1,954

Re: [VB6] - Module for working with multithreading.

have you worked out using a typelib that is compiled into the .RES as well? is the manifest able to point to an embedded typelib?
is it as simple as modifying the xml to something like this?

Code:
```
<file name="MarshalUserInterface.exe/3">
  <typelib tlbid="{2E8B35BD-EE5B-4CB8-9EBB-132017779212}"
           version="1.0"
           helpdir=""/>
</file>
```

Imagine what it would be like to set breakpoints in, or step through subclassing code;
and then being able to hit stop/end/debug or continue, without crashing the IDE.
VB6.tlb | Bulletproof Subclassing in the IDE (no thunks/assembly/DEP issues)

(rate this post) **Reply** **Reply With Quote**

Yesterday, 09:21 AM #5

The trick
Thread Starter
Frenzied Member
Join Date: Feb 2015
Posts: 1,045

Re: [VB6] - Module for working with multithreading.

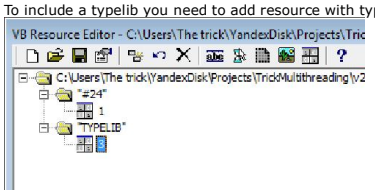
xxdoc123, DEXWERX thanks for review!

> Originally Posted by **DEXWERX**
> *is the manifest able to point to an embedded typelib?*
> *is it as simple as modifying the xml to something like this?*

Yes. You can specify the embedded typelib too.

Added.

To include a typelib you need to add resource with type "typelib" and needed identifier:



You need to specify the typelib as:

Code:
```
<file name="MarshalUserInterface.exe\3">
  <typelib tlbid="{2E8B35BD-EE5B-4CB8-9EBB-132017779212}"
           version="1.0"
           helpdir=""/>
</file>
```

with backslash.

Last edited by The trick; Yesterday at 09:35 AM.

EXE loader|Inline assembler Add-in|Kernel-mode driver on VB6|Multithreading in StandartEXE|Native VB6-DLL
Code injection to other process|DLL injection|DirectX9|DirectSound|TrickControls|MP3 player from memory
Easy calling by pointer|Hash-table|Safe subclassing|Vocoder|Creation of native DLL|COM-DLL without registration
FM-synthesizer|FFT spectrum-analyser|3D Fir-tree|Asynch waiter|Wave steganography|Fast AVI-trimmer
Windows with custom rendering|String to integer and vice versa|Multithreading with marshaling|Owner-draw listbox mod
Advanced math|Library info|Create GIF-animation|Store data to self-EXE|Computer creates a music|TrickSound class

(rate this post) **Reply** **Reply With Quote**

Yesterday, 09:36 AM #6

DEXWERX
Frenzied Member
Join Date: Jun 2015
Posts: 1,954

Re: [VB6] - Module for working with multithreading.

Thanks! I can't think of a use case you haven't covered.

Imagine what it would be like to set breakpoints in, or step through subclassing code;
and then being able to hit stop/end/debug or continue, without crashing the IDE.

VB6.tlb | Bulletproof Subclassing in the IDE (no thunks/assembly/DEP issues)

(rate this post) Reply Reply With Quote

+ Reply to Thread Quick Navigation CodeBank - Visual Basic 6 and earlier Top

Quick Reply

Post Quick Reply Go Advanced Cancel

« Previous Thread | Next Thread »

VBForums VBForums CodeBank CodeBank - Visual Basic 6 and earlier [VB6] - Module for working with multithreading.

Thread Information

There are currently 1 users browsing this thread. (1 members and 0 guests)

dz32

Tags for this Thread

multithreading, threading

View Tag Cloud

Posting Permissions

You may post new threads	**BB code** is On
You may post replies	**Smilies** are On
You may post attachments	**[IMG]** code is On
You may edit your posts	**[VIDEO]** code is On
	HTML code is Off
	Forum Rules

Contact Us VB Forums Top

Acceptable Use Policy

All times are GMT -4. The time now is 10:24 PM.