Knitting Your Own Threads

Presenting a better model for multithreaded applications

by Hallvard Vassbotn

All the Win32 platforms (Win95, Win98 and WinNT) support the concept of threading. This allows you to split a process into several independent execution threads, each with its own local stack and instruction pointer. This is a powerful concept that can improve the responsiveness of the user-interface, run useful code while waiting for I/O operations to complete, and improve the overall performance of your application (in the case of WinNT running on a multiprocessor machine).

Delphi has supported writing multithreaded code ever since the first 32-bit version, Delphi 2. There have been some thread-related articles in *The Delphi Magazine* before. In his *Under Construction* column in Issue 20 (April 1997), *Multi-Threading In Components*, Bob Swart gives a good introduction on multithreading in general and how to use the `TThread` class. In addition, back in Issue 17 (January 1997), Neil McClements showed us how to do *Multi-Threaded Database Access With Delphi 2.0*.

I will assume you already know how multithreading works. In the following sections, we will briefly see how the multithreading support in Delphi is built up. Then we will continue with how we can build upon this foundation to improve the usefulness of multithreading in Delphi.

Threads In The Box

The Object Pascal Language itself has support for threads with the `threadvar` keyword. This can be used to declare global variables that will be unique for each thread running in the process. Behind the scenes this is implemented by using one slot of the TLS (thread local storage) facility of Win32. This slot contains a pointer to a record with all the `threadvars` in the process. For details of the implementation, look up `TlsAlloc` in the Win32 help. In `System` (Delphi 2) or `SysInit` (Delphi 3 and 4) you will find the following interfaced declarations to aid in the `threadvar` support:

```
var
  TlsIndex: Integer;
  TlsLast: Byte;
procedure _GetTls;
```

The first variable, `TlsIndex`, indicates the index of the TLS slot that the RTL is using for the process' `threadvar` variables. `TlsLast` is a magic variable that the linker sets up so that its address contains the combined size of all the `threadvar` variables. This size is then used to allocate a block of memory from the local heap for each running thread in the process.

The `_GetTls` procedure is a magic routine that the compiler calls whenever you reference a `threadvar` variable. This routine actually acts as a function and returns the pointer to the local heap block allocated for the current thread. If you intend to access `threadvar` variables from assembly code, you should call the `GetTLS` function explicitly. To get at a specific variable within the returned record pointer, you simply use the dot (.) qualifier. For instance, to access the `InOutRes threadvar` from assembly code, you can write:

```
CALL    SysInit.@GetTLS
MOV     EAX,[EAX].InOutRes
```

The value of the `InOutRes` variable for the current thread will now be in the `EAX` register. Note that this code will work in Delphi 2 and 3, but will not compile in Delphi 4, because the variable has been moved out of the interface part and into the implementation part of the `System` unit. Use the `IOResult` function to access the contents of the variable.

`Threadvars` are best used in low-level and system-wide services such as the `InOutRes` example above. In fact, in all of the RTL only two `threadvars` are used: `RaiseList` and `InOutRes`. For instance, if you have a library that returns error codes through a single global variable, you might have to convert it into a `threadvar` to make the code support multiple threads.

In most other situations, you can safely forget about `threadvars`. It is better to use the fields within a `TThread` instance as the per-thread working area.

The System Is Building Up

The next level of support for threads can be found in the `System` unit. All 32-bit versions of Delphi (2, 3 and 4) have the Listing 1 code declared.

This variable was added in Delphi 4:

```
var
  MainThreadID: LongWord;
```

`IsMultiThread` is a global variable indicating if there is more than one thread running in the process. This is checked in all the routines of the memory sub-allocator and

➤ *Listing 1*

```
var
  IsMultiThread: Boolean;
type
  TThreadFunc = function(Parameter: Pointer): Integer;
function BeginThread(SecurityAttributes: Pointer; StackSize: Integer;
  ThreadFunc: TThreadFunc; Parameter: Pointer;
  CreationFlags: Integer; var ThreadId: Integer): Integer;
procedure EndThread(ExitCode: Integer);
```

ensures thread-safe operation while keeping the overhead at a minimum for single-threaded applications.

The IsMultiThread flag is set to True in a few places. BeginThread sets IsMultiThread to True when it creates a new thread in the application. This routine is used internally by the TThread class, so this covers the cases when you call BeginThread directly or use TThreads. If you for some reason need to call the Windows.CreateThread API directly, you should set IsMultiThread to True explicitly to avoid re-entrancy problems in the memory manager.

IsMultiThread was also set to True in DelphiMM library in Delphi 2 and 3. There is no real need to do this unconditionally, but I guess Inprise (Borland at the time) played it safe in case the DLL(s) or application contained multiple threads. In Delphi 4 this has changed a bit. The DelphiMM library is now only a stub that redirects all calls to the BorlandMM.DLL. This DLL is not provided with any source code, so it is not known how it is implemented with regard to the IsMultiThread variable. The new version of the ShareMem unit now imports from the new BorlandMM.DLL instead of the old DelphiMM.DLL.

In Delphi4, COM support has now also been made optionally thread-safe. If you select that the Apartment-Threaded or Multi-Threaded model for your COM objects, the ComObj unit will now kindly set the IsMultiThread flag to True for us.

In your own code, if you need to conditionally support both single- and multi-threaded operations, you can check the IsMultiThread variable. For instance, if you don't want to lock a resource if there is only one thread running in the process, but still want to make sure the code is safe when it is called from multiple threads, use the code in Listing 2.

The BeginThread and EndThread routines are thin wrappers around the corresponding Win32 APIs CreateThread and ExitThread. You can look these up in the Win32 help file for more detailed information. BeginThread adds the convenience of initialising the FPU and setting up and removing a default exception handler for the thread.

```
if IsMultiThread then EnterCriticalSection(MyLock);
try
  // Access and update global resource here
finally
  if IsMultiThread then LeaveCriticalSection(MyLock);
end;
```

➤ *Listing 2*

```
unit Unit1;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
  protected
    procedure Execute; override;
  end;
implementation
{ Important: Methods and properties of objects in VCL can only be used in a
  method called using Synchronize, for example, Synchronize(UpdateCaption);
  and UpdateCaption could look like:
    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

➤ *Listing 3*

BeginThread is called to create a thread that runs a specific procedure with the signature:

```
TThreadFunc = function(
  Parameter: Pointer): Integer;
```

This can be handy when you want to run a global routine in a thread. However, it is generally cleaner and more useful to use a TThread.

In Delphi 4 the MainThreadID variable was added. This is initialised by a call to the Win32 API GetCurrentThreadID. This lets you easily get at the threadID of the main thread in the process. This can be useful if you need to use a Win32 API that requires a threadID, for instance PostThreadMessage.

Let's Get Classy

With the low level support of the compiler and RTL out of the way, let's step up to the next level of abstraction. The Classes unit declares the TThread class, this is the base class of all thread classes in a process. It is basically a friendly wrapper around the BeginThread routine and provides simple-to-use properties and routines for setting the priority of the thread, suspending and resuming the thread, and so on. All the details are in the help file.

TThread is an abstract class because it does not define any implementation for the protected method called Execute. So you create a new thread class that descends from TThread and override the Execute method to do whatever processing is needed. For instance, running the New | Thread Object wizard generates the code in Listing 3.

As you can see from the generated comments in this code, most of the VCL is not thread-safe, so you must use the Synchronize mechanism to update the state of components and so on. This forces the method to run in the context of the main thread. While the synchronised code is run, the worker thread is suspended. This defies much of the reason to use multiple threads in the first place and this restriction is one of the major drawbacks with the standard multithreading support in Delphi.

Another problem with this model is that it breaks encapsulation. The working thread has to keep pointers to the form or components that should be updated. In addition, the idea of creating one thread for each background task, might not always be the best solution.

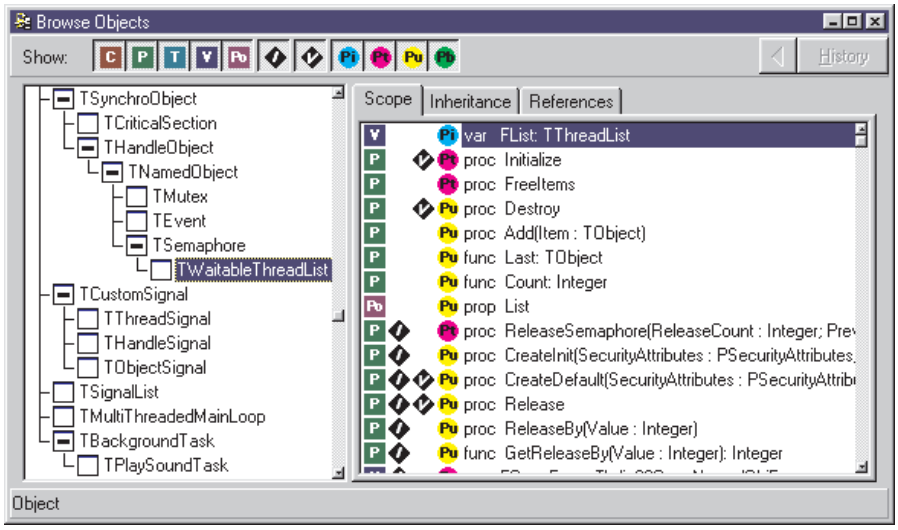
The `TThread` class also defines a `WaitFor` method. This can be called by another thread, it will block until the working thread is done. If we call `WaitFor` on a thread from the main thread, for instance, this will freeze the user interface that is controlled and painted by the main thread. Again, this defies the reason for writing multithreaded code. Also, there is no direct support for waiting for more than one thread at a time.

With all this said, `TThread` is still a nice encapsulation of the thread concept. We will later see how we can work around most of these restrictions and create a better infrastructure for multithreaded applications.

Add Some Sugar

When you create professional multithreaded applications, you will need support from some more basic building blocks in addition to the `TThread` class. For instance, `Classes` defines the `TThreadList`, which is a thread-safe version of the well known `TList` workhorse. It is basically just a wrapper around a normal `TList` and an associated Win32 API `TRTLCriticalSection` that provides the locking capability of the list.

The `SyncObjs` unit was introduced in Delphi 3 and provides thin wrappers around the useful Win32 objects `Events` and `CriticalSections`. In Delphi 4 they added a class with the very exotic name `TMultiReadExclusiveWriteSynchronizer` to the `SysUtils` unit. This class is very useful when you have a large number of threads that need to access a common resource, and most threads need read access. It allows any number of threads simultaneous read access, while locking out all other threads when one thread needs write access.



➤ *Figure 1*

Nevertheless, there are still a lot of things missing. There are no object wrappers around semaphores or mutexes, no wrapping of the `WaitForMultipleObjects` API, no way to wait for signalled objects in the main thread without blocking the user interface. I could go on. The point is that the multithreading capabilities still leave things to be desired. The good news is that we can do something about it.

Making A Better Mouse-Trap

I have extended the basic support for synchronisation objects found in `SyncObjs`. Take a look at the `HVSyncObjs` unit provided on this month's disk. As well as the existing `TEvent` and `TCriticalSection`, I have added `TMutex`, `TSemaphore`, and a higher level object called `TWaitableThreadList`.

All these classes descend from `TSyncroObject` to allow for polymorphic calls to the `Acquire` and `Release` methods. This allows us to write code that will work equally well with a low-overhead critical section as with an inter-process mutex object. The implementation code does not have to change and the decision of what class to use could even be made at runtime.

All the waitable classes (all but `TCriticalSection`) descend indirectly from `THandleObject`. This allows for polymorphic waiting for these objects. They also descend directly from `TNamedObject`. This gives the classes a set of constructors to create named and anonymous objects as well as open existing objects created in an external process.

Finally, there is the `TWaitableThreadList` class. This descends from `TSemaphore` and adds an encapsulation of a thread-safe list. This implementation lets us wait until one or more items have been added to the list by a secondary thread. This lays the ground for a thread-safe and efficient communication mechanism between threads.

The hierarchy of the classes in the `HVSyncObjs` unit is in Figure 1.

The design of `HVSyncObjs` is open-ended so you can easily add more synchronisation classes to your arsenal. For instance, I/O Completion ports as supported by Windows NT could be worthwhile using for server-side applications to reduce the number of threads needed to service client requests.

The implementation of the `TCriticalSection`, `TMutex`, `TEvent` and `TSemaphore` classes is straightforward: thin wrappers around the corresponding Win32 API routines and structures. We can summarise the typical uses of each of these classes in Table 1.

If you need more detailed information on how to use these classes, look up the corresponding Win32 API functions in the help.

Waiting For Godot

The most useful class in the `HVSyncObjs` unit is without doubt

Class	Description And Typical Uses
TCriticalSection	Used to lock a common global resource (memory, file, object etc). Cannot be used between two processes.
TMutex	Used to lock a common global resource. Can be used between two processes.
TEvent	Signal the occurrence of some event to one or more threads. Can be used between two processes.
TSemaphore	Allows n-number of threads access to a resource. Can also be used as a counting semaphore to keep track of the number of items added to a list, for instance. Can be used between two processes.

➤ *Table 1*

Operation	Description
Add	Add an item to the beginning of the list in a thread-safe manner. This will also release the semaphore, signalling to any other thread that might be waiting for this event.
WaitFor	Wait for the semaphore object to become signalled. This will happen when some other thread calls the Add method.
Last	Retrieve the last item from the list. This should only be called from a thread that received a signal from the list after a call to WaitFor (or equivalents).
Count	Query the number of items currently in the list.

➤ *Table 2*

the `TWaitableThreadList`. This class will be used as the basis for the improved thread communication mechanism we are going to need later. The `TWaitableThreadList` class only supports four operations besides creation and destruction, see Table 2.

So typically, one thread keeps calling the `Add` method at random intervals, while another thread blocks on a call to `WaitFor`, wakes up and then calls the `Last` method to see what the other thread added to the communication line. The list has semantics like a `FIFO` (first in, first out) queue.

This class can be used to administer a background working thread. The thread class will keep two `TWaitableThreadLists`, one `InBox` for work to be done and one `OutBox` for worked finished by the thread ready to be picked up by the main thread (or even another work thread for further processing), see example work flow in Figure 2.

So far, so good. Now we have a reasonably clean and efficient way of communicating between the main thread and a working thread. This avoids using the problematic `Syncronize` method of `TThread`. However, we still have not solved the issue of waiting for multiple events from the main thread.

The MainThread Dilemma

The problem is that the main thread needs to service any pending messages in its message queue in addition to the events signalled by our working thread. During normal operation, when the application goes idle and there are no more messages in the message queue, the main thread will block with a call to `WaitMessage` inside `TApplication.Idle`. As soon as a message is added to the queue, the main thread will wake up to handle the message.

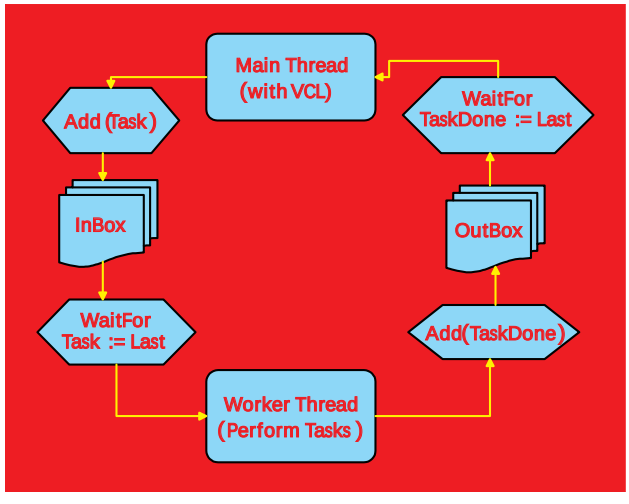
So how can we change this arrangement so that the main thread will not only wake up on the arrival of a message, but also on the signalling of any number of events or semaphores? It is not recommended to modify the source of `TApplication` inside the `Forms` unit and this would not work at all when using runtime packages or for developers without the VCL source code.

It turns out that we can solve this problem by hooking into the `OnIdle` event of `TApplication`. Inside our `OnIdle` event handler, we will call the powerful Win32 API called `MsgWaitForMultipleObjects`. This routine is able to block the execution of the thread until a message arrives at the message queue or any of a number of objects becomes signalled. This is just what we need. If we wake up because of a message, we immediately return to `TApplication.Idle` to let it handle the message. Otherwise, we handle the signal from the object and loop around to call `MsgWaitForMultipleObjects` again.

Yet Another Wrapper

Instead of calling `MsgWaitForMultipleObjects` directly, we want to write another wrapper class for it. This should gives us a nicer interface and let us dynamically add any number of threads, handles or `THandleObjects` to be waited for. In the `HVSignalList` unit, you will find the implementation of the `TSignalList` class. The two most important methods of this class can be found in Table 3.

➤ *Figure 2*



There are other methods and properties to determine how the SignalList should operate. For instance, we can specify what kind of messages should wake up the thread (MsgWakeupMask), if messages should be ignored altogether (IgnoreMessages), and if all objects must be signalled simultaneously for it to trigger (WaitForAll). See the source code for details.

In Table 3, we say that the AddSignal can be sent a TThread, THandleObject or THandle. To handle all cases in a generic manner and to encapsulate these different types of signalling objects, I've created an abstract class called TCustomSignal. This class has three descendant classes, TThreadSignal, TObjectSignal and THandleSignal. The AddSignal method takes a parameter of type TCustomSignal, but at runtime one of the three descendant classes will actually be used.

After you have added a TCustomSignal instance to the SignalList, the list takes ownership of the object. To add a TWaitableThreadList to the signal list we write code such as Listing 4.

Here we create the waitable list and the signal list, then we connect the InBox list to the SignalList by wrapping it up in a TObjectSignal and sending it off to the AddSignal method.

Notice the second parameter of the TObjectSignal.CreateInit constructor. It is a method pointer that will be called when the InBox object becomes signalled. This indicates that some other thread has added one or more items to our InBox and we should perform some relevant action (typically, we will call InBox.Last to examine the item just added).

Method	Description
AddSignal	Add a signalling object to the list of items that should be triggered on. This can be a TThread, a THandleObject or a raw Win32 THandle.
WaitUntil	Wait until one of the objects in the signal list triggers, until there is a message in the message queue, or until a timeout occurs.

➤ *Table 3*

```
var
  InBox: TWaitableThreadList;
  SignalList: TSignalList;
begin
  InBox := TWaitableThreadList.CreateSimple;
  SignalList := TSignalList.Create;
  SignalList.AddSignal(TObjectSignal.CreateInit(InBox, Self.InBoxReady));
  ...
```

➤ *Listing 4*

```
procedure TMultiThreadedMainLoop.AppIdle(Sender: TObject; var Done: boolean);
// Whenever the application becomes idle, i.e. there are no messages in the
// message queue, this procedure is entered.
begin
  // The default case for the old idle event
  // handler should be that it is done processing
  Done := true;
  // Call any old idle event handler
  // - this could be extended with an idle hook chain
  if Assigned(FOldAppIdle) then
    FOldAppIdle(Sender, Done);
  // WaitUntil handles all signalled objects for the main thread
  if Done then
    // If the old idle event handler is done,
    // wait until there is a message for us (blocking)
    FSignalList.WaitUntil(INFINITE, [wrMessage])
  else
    // If the old idle event handler is not done yet,
    // just check for signalled objects or messages (non-blocking)
    FSignalList.WaitUntil(0 , [wrMessage, wrTimeOut]);
  // Always return Done=False to signal that the message
  // loop should go back here when it has read all messages
  Done := False;
  // Tell the timer-event that we have actually been idle
  FHasBeenIdle := true;
  // Now return to the message loop in TApplication and
  // let it have a look at the message for us
end;
```

➤ *Listing 5*

Hooking OnIdle

Now that we have a nice encapsulation of MsgWaitForMultipleObjects to work with, let us continue improving the threading support of the main thread (and thus the VCL).

As we discussed, this is accomplished by hooking the OnIdle event of TApplication. See the source code of the TMultiThreadedMainLoop class in the HVMultiThreadMain unit.

There are some implementation details we must take care of. First, we keep any old event already assigned to the OnIdle property before assigning it to our AppIdle method. This is a good general rule whenever you hook into an external event property at runtime, always keep the old value and chain to it in your event handler.

Then we have to tackle the issue of multiple message loops within the application. Remember that our AppIdle will only be called from the main message loop within TApplication. However, whenever a menu or modal dialog is active, other message loops are running and we will never get control. This will disable us from seeing any signalled objects until we get back to the main message loop (when the dialog or menu is dismissed).

To get around this problem, we simply create a TTimer object and program it to call our OnIdleTimer event roughly 10 times per second. You can fine-tune this resolution by setting the IdleTimerInterval property, but the default setting will usually be appropriate. In the OnIdleTimer we check that we haven't been idle for some time and then empty any pending signalled objects in the signal list.

Now to the key method of the TMultiThreadedMainLoop class, the

OnIdleTimer event handler, see Listing 5. Let's study the implementation details of this essential method in some detail.

First, we chain back to any old idle event handler that has been installed. If the old event handler indicates that it is done processing (or if there was no old idle event handler), we call the WaitUntil method of the signal list instance with an INFINITE timeout parameter and [wrMessage] as a wait return mask. This call will not return until there is a message in the message queue of the main thread.

If there was an old idle handler and it indicates that it is *not* done processing, we call the WaitUntil method with a 0 timeout parameter. The call returns immediately even if there is no message in the message queue. This is to ensure that the busy old idle handler will be called again in due time.

Note that in both of the WaitUntil calls, if any of the objects in the signal list signals, it will be silently handled and the correct call back event will automatically be called. This is handled deep down in the implementation of the TSignalList class by calling the Windows API MsgWaitForMultipleObjects (see the code on the disk for details).

After we return from the WaitUntil method, we set a flag to let the timer event handler know that we have been idle. Then we return to TApplication.Idle indicating that we are not done processing by setting the Done parameter to False. If there was a message in the message queue, it will be picked up by the normal message loop of TApplication.

```
procedure TApplication.Idle(const Msg: TMsg);
var Done: Boolean;
begin
  ...
  Done := True;
  if Assigned(FOnIdle) then
    FOnIdle(Self, Done);
  if Done and IsIdleMessage(Msg) then
    DoActionIdle;
  if Done then
     WaitMessage;
end;
```

➤ *Listing 6*

The Delphi 4 Blues

It turns out that this logic works beautifully in Delphi 2 and 3. It also works in Delphi 4, but has a rather disconcerting side-effect: the new action list based functionality ceases to work correctly. Specifically, the OnUpdate event of the embedded TAction components no longer fires. This causes any controls linked to the actions to stop updating their enabled state properly, for instance.

The reason for this problem was easy to find. The Idle method of TApplication was changed in Delphi 4 to accommodate the new action lists and their automatic ability to determine if command controls (such as buttons and menu items) should be updated.

These few lines from the TApplication.Idle method in Delphi 4 explains why this happens (I have deleted code that does not affect our case). The if statement ending with the DoActionIdle call is new for Delphi 4, see Listing 6

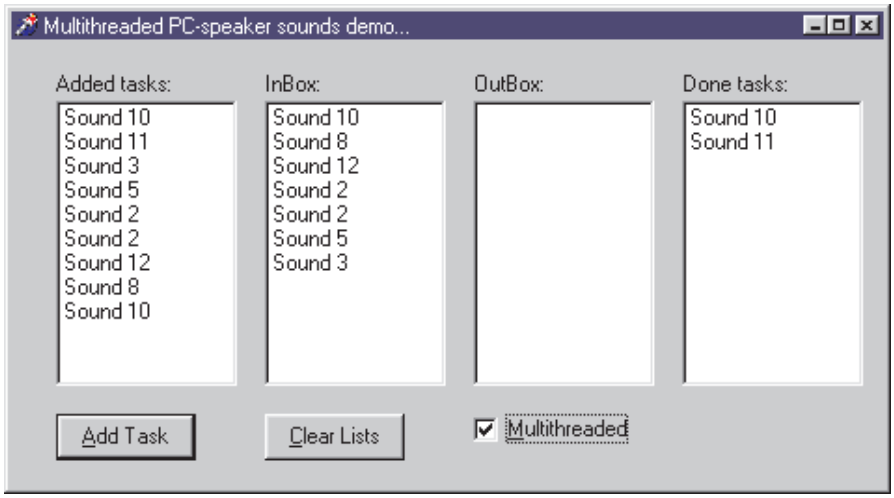
We are trapped in a Catch-22 situation here. Our OnIdle handler will always return with Done equal to False. This is to ensure that the blocking WaitMessage call inside TApplication.Idle is never made. The trouble is that this has the side-effect of never calling the DoActionIdle method. If we change our code to return with Done set to True, the action list functionality should be back on its feet, but our signalling objects will be left in the dark until some message happens to arrive in our message queue.

Saved By The Bell

When writing this article, I was just about to end up in the trap of trying to work around this problem by copying the logic of DoActionIdle in my own code. This would have worked, but it would have been a very ugly, version-specific and unmaintainable solution. In fact, I had written the code and several paragraphs explaining it and its drawbacks, when I suddenly realised that there is a much simpler and more elegant solution.

All along I have assumed that Done *must* be False when we return from our OnIdle handler. After all, this is what you normally have to do in OnIdle handlers to make certain we get control back immediately after handling any pending messages. However, we are not talking about your run-of-the-mill OnIdle handler here. We are in the special situation that we only return to TApplication.Idle in the event where we *know* that there is at least one message pending in the message queue (because TSignalList.WaitUntil returned wrMessage). We could also return without any messages in the message queue, but then the Done parameter would have been set to True by the old idle event handler.

➤ *Figure 3: The sample multithreaded application.*



With this knowledge, there is actually no harm in returning with `Done` equal to `True`. This will call `WaitMessage`, but it will *not* block because, there is always a message waiting. So the solution is simply to remove the `Done := False` assignment from Listing 5.

Now, with that analysis behind us, this simple change solves the Delphi 4 problem quite elegantly. The `DoActionIdle` will be correctly called and the `ActionLists` should again behave as expected. Furthermore, our code is now the same for all versions of Delphi, and it should be safe in the case of future changes in the implementation of `TApplication.Idle` (unless Inprise starts using `MsgWaitForMultipleObjects` themselves...).

Demo Project

To demonstrate some of the new units and classes we have been discussing, I have provided a simple demo project on the disk. It plays sounds through the PC speaker (ancient technology, I know) in a background thread. It could just as well have performed lengthy calculations or searched through text files. See Figure 3.

Conclusion

With this set of tools, you should be much better equipped to write multithreaded applications. Your threaded code should be able to support better encapsulation of logic and let you isolate application specifics and user interface details from the working threads. The signal lists should give you a cleaner communication channel between threads and the multithreaded main loop class will finally let you wait on multiple objects while still handling the message queue properly.

Hallvard Vassbotn is a Senior Software Developer at Reuters Norge AS, Falcon R&D. You can reach him at hallvard@falcon.no