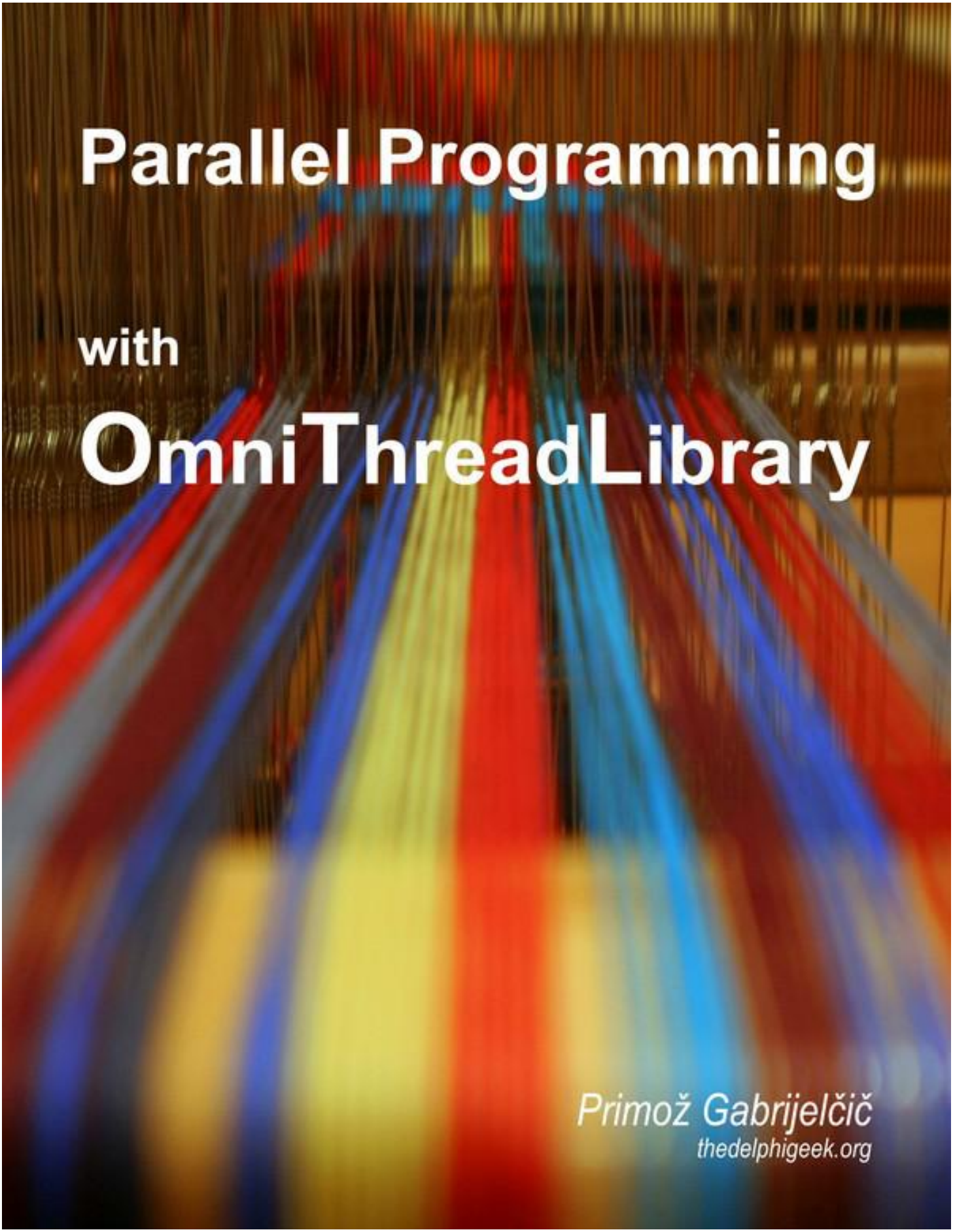

Parallel Programming
with
OmniThreadLibrary
Primož Gabrijelčič
thedelphigeek.org

1. Introduction to multithreading

You may have heard about threads and mutltihreading already, but you may not exactly know what those words actually mean. The precise definition is not important for this story. What is important are the general ideas. If you want to read more, you should start with the entry in Wikipedia. If you are well versed in these topics, you can freely skip ahead to the next chapter, Introduction to OTL.

When you start a program, the operating system creates internal entity called a process. It contains almost everything that an operating system associates with the started program - allocated memory, open file and window handles, sockets and so on. A process, however, does **not** contain the information related to the CPU state. That is part of a thread.

A thread encapsulates the state of the CPU registers, thread stack and thread local variables. Every process contains one thread - the main thread. Operating system creates it together with a process when it starts a program.

Modern operating systems allow multiple threads of execution inside a process. This is achieved by creating additional background threads in the code. To create a thread we can use operating system primitives (such as `CreateThread` on Windows), low-level runtime library functionality as Delphi's `TThread`, or higher-level concepts such as tasks and abstractions (also known as patterns).

On a multi-CPU machine these threads can execute in parallel, one on each core. A typical computer system, however, runs much more threads than it has CPUs and that's where multithreading kicks in.

A multithreading is not a new invention. It appeared decades before multiprocessor boards became available to the general public. In a typical multithreading system the operating system allows a thread to run for some time (typically few milliseconds) and then stores the thread's state (registers etc) into the operating system's internal data structures and activates some other thread. By quickly switching between multiple threads the system gives an illusion that it is running many threads in many programs in parallel.

1.1 Multithreading as a source of problems

Multithreading is a powerful concept, but it also brings many problems. They are all consequence of the same fact – that threads are not isolated.

An operating system works hard to isolate one process from another. This prevents one process from accessing the memory belonging to another, which is an important security feature. It also makes sure that a bug in one process won't affect any other process in the system. If you think that's nothing special, you are not old enough. Some of us had to work on Windows 3 which had no such isolation and we still remember how one badly behaved application could bring down the whole system.

Since threads are not isolated, each thread can access all of the memory memory belonging to a process. This gives us extreme power and flexibility. Starting multiple threads provides all of the threads with access to the same data. It is also a source of all problems. If multiple threads don't only read from the shared data but write to it, you'll get into trouble.

It should be obvious that one thread must not modify data while another thread is reading from it. It is harder to enforce this in practice. We can protect access to shared data by

using locking (critical sections, spinlocks, TMonitor ...) or interlocked operations, but they will slow the program down and they can introduce new problems (deadlocks).

That is why I prefer data copying, communications and aggregation over data sharing. Modern CPUs are fast and they can easily copy few tens or hundreds of bytes that are then sent to the thread so it can work on its own copy of the data and not on shared data. This approach is the basis of OmniThreadLibrary and you'll see it all through the book.

To write better code you should know which operations that are completely safe in single-threaded world will get you in trouble when you are multithreading.

1.1.1 Reading and writing shared data

Let's start with a simple variable access. One thread is reading from a variable and another is writing to it.

```
1 var
2   a,b: integer;
3
4 // thread 1
5
6 a := 42;
7
8 // thread 2
9
10 b := a;
```

This looks safe. Variable b will contain either 42 or previous data stored in a. Well, that is not entirely true. b may also contain a mixture of both.

Depending on how the a is stored in memory (how it is aligned) the processor will access it either with one or two operations. If two operations are used, a read operation may partially overlap the write operation. For example, the code may read the first part of a, then a gets fully updated and only then the code reads the second part of a.

This code behaves "as expected" (doesn't cause problems) if the address of a gives remainder of 0 when divided by 4. We say that a is 4-aligned. Delphi in most cases makes all integers 4-aligned. Sadly, in some occasions, especially when structured types – classes and records – are nested, variables are not well-aligned.

We say that the operation is atomic if it is performed in one step. Reads and writes of 4-aligned integers are atomic on Intel platform.

The simplest way to be sure that a value is correctly aligned is to use the TOmniAlignedInt32 record provided by the OmniThreadLibrary. It makes sure that the underlying data storage is correctly aligned and can as such be accessed atomically.

OmniThreadLibrary also implements TOmniAlignedInt64 type which creates 8-aligned int64 data. This data may be accessed atomically only from 64-bit code.

Reads and writes of 8-aligned int64s are atomic only in 64-bit code.

TOmniAlignedInt64 implements atomic operations on int64s that can be used in 32-bit code.

1.1.2 Modifying shared data

Another example of typical code that doesn't work correctly in multithreaded environment is modifying the same data from two threads. In a typical program this data

modification is hidden inside an operation that we think of as atomic while in reality it isn't.

A standard example of this problem is implemented with two threads, one incrementing the shared value and other decrementing it.

```
var
  data: integer;

data := 0;

// thread 1

for i := 1 to 100000 do
  Inc(data);

// thread 2

for i := 1 to 100000 do
  Dec(data);
```

In a single-threaded world, the value of `data` is 0 when this pseudo code finishes. In a multithreaded scenario, however, it will lie somewhere between -100000 and 100000 – and that's all we can say.

The problem with this code is that `Inc` and `Dec` are not atomic operations. Rather they are implemented inside the CPU as a series of three operations – read data from memory, increment (or decrement) data, and write data to memory.

One way to fix such code is to use already mentioned `TOmniAlignedInt32` which implements atomic `Increment` and `Decrement` operations.

```
var
  data: TOmniAlignedInt32;

data := 0;

// thread 1

for i := 1 to 100000 do
  data.Increment;

// thread 2

for i := 1 to 100000 do
  data.Decrement;
```

Another option is to use interlocked operations from Delphi's System.SyncObjs unit.

```
var
  data: integer;

data := 0;

// thread 1

for i := 1 to 100000 do
  TInterlocked.Increment(data);

// thread b

for i := 1 to 100000 do
  TInterlocked.Decrement(data);
```

The third option is to protect increment/decrement operations by wrapping them in a

critical section. You could, for example, use OmniThreadLibrary's TOmniCS.

```
var
  data: integer;
  lock: TOmniCS;

data := 0;

// thread 1

for i := 1 to 100000 do begin
  lock.Acquire;
  data.Increment(data);
  lock.Release;
end;

// thread 2

for i := 1 to 100000 do begin
  lock.Acquire;
  data.Decrement(data);
  lock.Release;
end;
```

Using interlocked operations is a bit slower than normal arithmetics and locking with critical sections is even slower. That's one of the reasons I prefer to use data copying and aggregation.

The next example shows how the increment/decrement example could be rewritten with these principles in mind.

```
var
  data: integer;

// thread 1
var
  tempData: integer;

tempData := 0;
for i := 1 to 100000 do
  Inc(tempData);

send tempData to main thread

// thread 2
var
  tempData: integer;

tempData := 0;
for i := 1 to 100000 do
  Dec(tempData);

send tempData to main thread

// main thread

data := result from thread 1 + result from thread 2
```

This approach doesn't manipulate shared data which makes it run faster than interlocked and locking solutions. It does, however, require some communication mechanism that will send data to and from thread. Different parts of this book will deal with that.

1.1.3 Writes masquerading as reads

Another source of problems are shared objects. If we share an object between threads and then execute some methods of that object in different threads we have to know exactly

whether these operations are all thread-safe (that is, whether they can be executed in parallel from multiple threads).

A simple example, which I saw in real code, shares a stream between two threads. The worker thread is reading from the stream and doing something with the data (it doesn't matter what). The main thread is monitoring the progress by tracking stream's `Position` and updating a progress bar on the user interface.

In pseudocode, these two threads are executing following operations on a shared stream.

```
var
  data: TStream;

// worker thread

data.Read(...);

// main thread

UpdatePosition(data.Position / data.Size);
```

The problem here is easy to overlook as we percieve all operations on the shared stream as reading. In reality, this is not so. Let's at them in more detail.

`Read` reads data from stream's current position and then **updates** that current position.

`Position` just reads the current position.

`Size` is the worst of them all. Internally it is implemented as three `Seek` operations.

```
  Pos := Seek(0, soCurrent);
  Result := Seek(0, soEnd);
  Seek(Pos, soBeginning);
```

The problem here is that `Seek` **modifies** the current position. By calling `Size` in one thread we can change the current position just as the other thread starts reading from the stream. That will cause wrong data to be read from the stream.

The morale here is simple. Sharing data between thread is **always** dangerous.

2. Introduction to OmniThreadLibrary

OmniThreadLibrary is a multithreading library for Delphi, written mostly by the author of this book (see Credits for full list of contributors). OmniThreadLibrary can be roughly divided into three parts. Firstly, there are building blocks that can be used either with the OmniThreadLibrary threading helpers or with any other threading approach (f.i. with Delphi's `TThread` or with AsyncCalls). Most of these building blocks are described in chapter Miscellaneous, while some parts are covered elsewhere in the book (Lock-free Collections, Blocking collection, Synchronization).

Secondly, OmniThreadLibrary brings low-level multithreading framework, which can be thought of as a scaffolding that wraps the TThread class. This framework simplifies passing messages to and from the background threads, starting background tasks, using thread pools and more. In some ways it is similar to `ITask` which was introduced in Delphi XE7 except that OmniThreadLibrary's implementation offers more rounded feature set.

Thirdly, OmniThreadLibrary introduces high-level multithreading concept. High-level framework contains multiple pre-packaged solutions (so-called abstractions) which can be used in your code. The idea is that the user should just choose appropriate abstraction (Future, Pipeline, Map ...) and write the worker code, while the OmniThreadLibrary provides the framework that implements the tricky multithreaded parts, takes care of synchronisation and handles other menial tasks.

2.1 Requirements

OmniThreadLibrary requires at least Delphi 2007 and doesn't work with FreePascal. The reason for this is that most parts of OmniThreadLibrary use language constructs that are not yet supported by the FreePascal compiler.

High-level multithreading framework requires at least Delphi 2009. Delphi XE or newer is recommended as some parts of the framwork aren't supported in Delphi 2009 and 2010 due to compiler bugs.

OmniThreadLibrary currently only works in Windows applications. Both 32-bit and 64-bit platform are supported. Applicatins can be compiled with the VCL library, as a service or as a console application. FireMonkey is currently not supported.

2.2 License

OmniThreadLibrary is an open-sourced library with the OpenBSD license.

> This software is distributed under the BSD license.
>
> Copyright (c) Primoz Gabrijelcic
>
> All rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation

and/or other materials provided with the distribution.
- The name of the Primoz Gabrijelcic may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

In short, this means that:

1. You can use the library in any project, free, open source or commercial, without having to mention my name or the name of the library anywhere in your project, documentation or on the web site.
2. You can change the source for your own use. You can also put a modified version on the web, but you must not remove my name or the license from the source code.
3. I'm not guilty if the software blows in your face. Remember, you got OmniThreadLibrary for free.

In case your company would like to get support contract for the OmniThreadLibrary, please contact me.

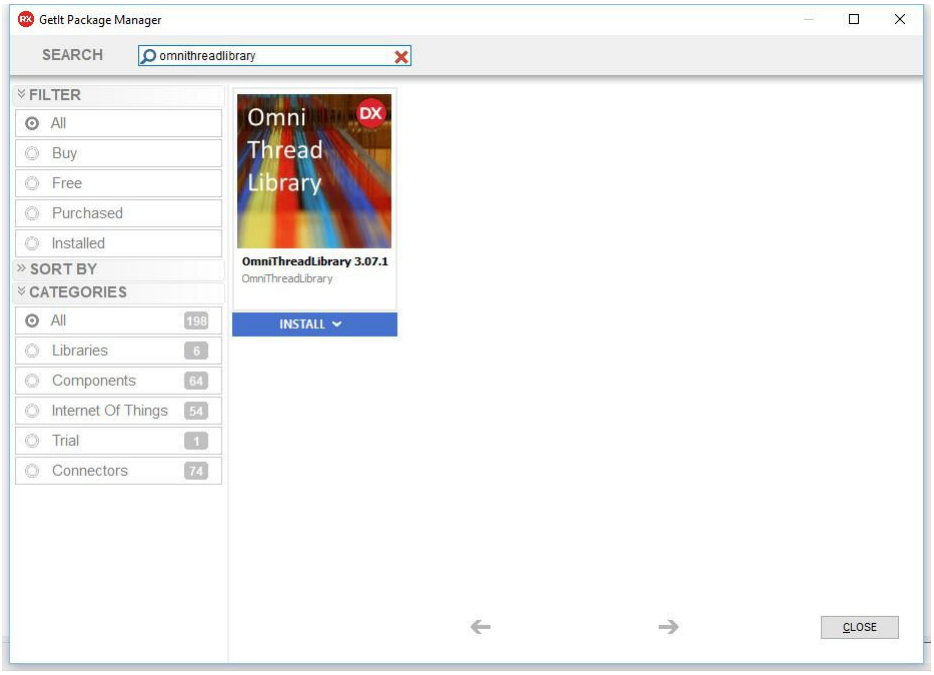
2.3 Installation

1. Download the last stable edition (download link is available at the OmniThreadLibrary site), or download the latest state from the repository. Typically, it is safe to follow the repository trunk as only tested code is committed.
2. If you have downloaded the last stable edition, unpack it to a folder.
3. Add the folder where you [unpacked last stable edition/checked out the SVN trunk] to the Delphi's Library path. Also add the src subfolder to the Library path. In case you are already using units from my GpDelphiUnits project, you can ignore the copies in the src folder and use GpDelphiUnits version.
4. Add necessary units to the `uses` statement and start using the library!

2.3.1 Installing with GetIt

Delphi/RAD Studio XE8 and newer come with an integrated package manager called GetIt.

With GetIt you can install OmniThreadLibrary with just a few clicks. Start Delphi and open **Tools, GetIt Package Manager**. Enter 'omnithreadlibrary' into the search bar. Then click on the **INSTALL** button under the OmniThreadLibrary graphics.



GetIt will download OmniThreadLibrary from Embarcadero's servers, add source path to the Library path and compile and install the design package.

To find the demos, look at the **Library path**. It will contain something like this at the end: `$(BDSCatalogRepository)\OmniThreadLibrary_3.07.1-Tokyo\src\`. To find the true path, look into **Tools, Options, Environment Options, Environment Variables** where `BDSCatalogRepository` is defined.

Removing OmniThreadLibrary from Delphi is equally simple. Just open GetIt, select the **Installed** category and click the **UNINSTALL** button under the OmniThreadLibrary graphics.

2.3.2 Installing with Delphinus

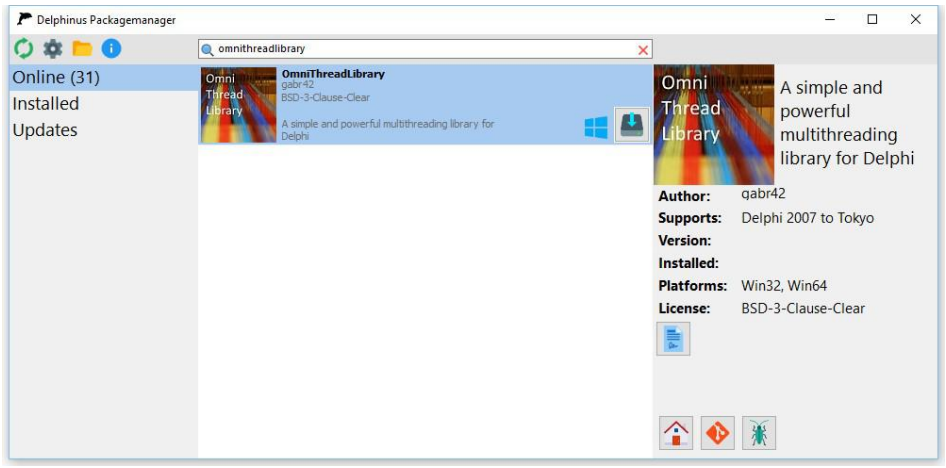
Delphinus is a 3rd party package manager for Delphi XE and newer, similar to Embarcadero's own GetIt package manager.

Unlike GetIt, which comes integrated into Delphi, you have to install Delphinus manually. Recommended installation procedure is:

1. Stop Delphi/RAD Studio.
2. Install Delphinus with the web installer.
    1. It is recommended to right-click on the installer and select 'Run as administrator'. Otherwise Delphinus may not be able to create the installation folder.
3. Start Delphi/RAD Studio.
4. (Optional, but highly recommended) Open **Tools, Delphinus**, click the **Settings** icon and enter **OAuth-Token**. This will prevent frequent GitHub 'rate limitation' errors during operation. Instructions for generating the token can be found on the Delphinus wiki.

Once Delphinus is installed, click the **Tools, Delphinus** and in the Delphinus window click the green **Refresh** icon. When the package list is refreshed, enter 'omnithreadlibrary'

into the search bar and press <Enter>. Click on the OmniThreadLibrary list item to get additional description in the panel on the right.



To install OmniThreadLibrary, click the **Install** icon (blue arrow pointing downwards to the disk drive). Be patient as Delphinus may take some time without displaying any progress on the screen.

When installation is complete, click the **Show log** button and in the log find the path where OmniThreadLibrary was installed (look for **Adding libpathes** message). Inside that folder you'll also find all OmniThreadLibrary demos.

K> You can find this path in Delphi's **Library path** configuration setting.

Delphinus will compile and install appropriate package so everything is set up for you.

Removing OmniThreadLibrary from Delphi is equally simple. Just open Delphinus, select the **Installed** category, select OmniThreadLibrary and click the **Remove** icon (red circle with white X).

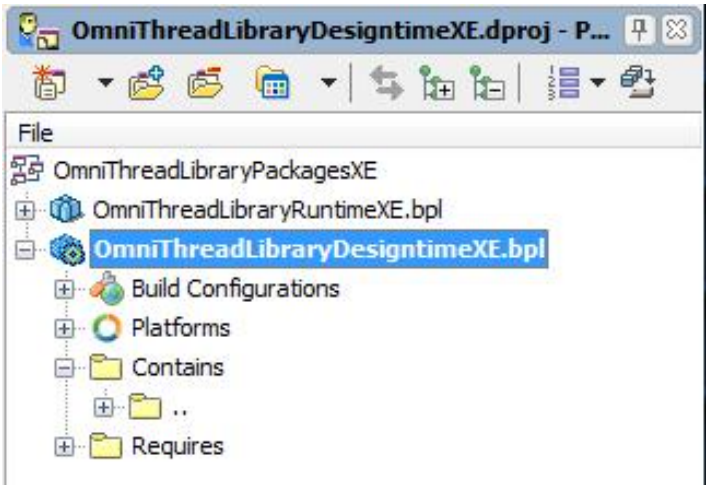
2.3.3 Installing design package

OmniThreadLibrary includes one design-time component (TOmniEventMonitor) which may be used to receive messages sent from the background tasks and to monitor thread creation/destruction. It is used in some of the demo applications.

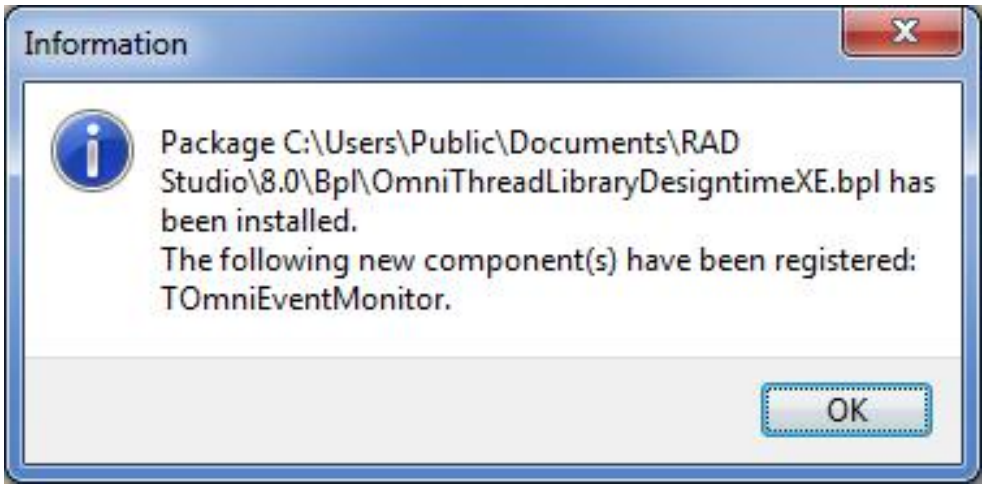
If you installed OmniThreadLibrary with GetIt or Delphinus the installation process has already installed the design package and you may omit this step.

To compile and install the package containing this component, follow these steps:

- From Delphi, open packages subfolder of the OmniThreadLibrary installation and select file OmniThreadLibraryPackages.groupproj from the appropriate folder.
- In the Project Manager window you'll find two projects – OmniThreadLibraryRuntime{VER}.bpl and OmniThreadLibraryDesigntime{VER}.bpl (where {VER} is **package version**[1] of your Delphi). If the Project Manager window is not visible, select View, Project Manager from the menu.



- Right-click on the OmniThreadLibraryRuntime{VER}.bpl and select Build from the pop-up menu.
- Right-click on the OmniThreadLibraryDesigntime{VER}.bpl and select Build from the pop-up menu.
- Right-click again on the OmniThreadLibraryDesigntime{VER}.bpl and select Install from the pop-up menu.
- Delphi will report that the `TOmniEventMonitor` component was installed.



- Close the project group with File, Close All. If Delphi asks you whether to save modified files, choose No.

You should repeat these steps whenever the OmniThreadLibrary installation is updated.

2.4 Why use OmniThreadLibrary?

OmniThreadLibrary approaches the threading problem from a different perspective than TThread. While the Delphi' s native approach is oriented towards creating and managing threads on a very low level, the main design guideline behind OmniThreadLibrary is: "Enable the programmer to work with threads in as fluent way as possible." The code should ideally relieve you from all burdens commonly associated with multithreading.

OmniThreadLibrary was designed to become a "VCL for multithreading" – a library that will make typical multithreading tasks really simple but still allow you to dig deeper and mess with the multithreading code at the operating system level. While still allowing this low-level tinkering, OmniThreadLibrary enables you to work on a higher level of abstraction most of the time.

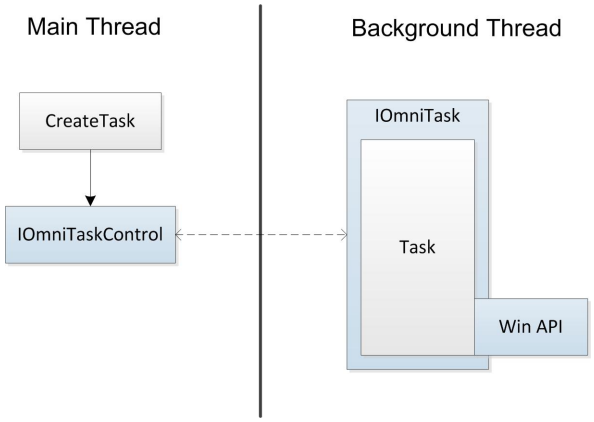
There are two important points of distinction between TThread and OmniThreadLibrary, both explained further in this chapter. One is that OmniThreadLibrary focuses on tasks, not threads and another is that in OmniThreadLibrary messaging tries to replace locking whenever possible.

By moving most of the critical multithreaded code into reusable components (classes and high-level abstractions), OmniThreadLibrary allows you to write better multithreaded code faster.

2.5 Tasks vs. threads

In OmniThreadLibrary you don't create threads but tasks. A task can be executed in a new thread or in an existing thread, taken from a thread pool.

A task is created using `CreateTask` function, which takes as a parameter a global procedure, a method, an instance of a `TOmniWorker` class (or, usually, a descendant of that class) or an anonymous method (in Delphi 2009 and newer). `CreateTask` returns an `IOmniTaskControl` interface, which can be used to control the task. A task is always created in suspended state and you have to call `Run` to activate it (or `Schedule` to run it in a thread pool).



The task has access to the `IOmniTask` interface and can use it to communicate with the owner (the part of the program that started the task). Both interfaces are explained in full detail in chapter Low-level multithreading.

The distinction between the task and the thread can be summarized in few simple words.

> Task is part of code that has to be executed.
>
> Thread is the execution environment.

You take care of the task, OmniThreadLibrary takes care of the thread.

2.6 Locking vs. messaging

I believe that locking is evil. It leads to slow code and deadlocks and is one of the main reasons for almost-working multithreaded code (especially when you use shared data and forget to lock it up). Because of that, OmniThreadLibrary tries to move as much away from the shared data approach as possible. Cooperation between threads is rather achieved with messaging.

If we compare shared data approach with messaging, both have good and bad sides. On the good side, shared data approach is fast because it doesn't move data around and is

less memory intensive as the data is kept only in one copy. On the bad side, locking must be used to access data which leads to bad scaling (slowdowns when many threads are accessing the data), deadlocks and livelocks.

To see how easy it is to run into problems with locking, play the The Deadlock Empire game.

The situation is almost reversed for messaging. There's no shared data so no locking, which makes the program faster, more scalable and less prone to fall in the deadlocking trap. (Livelocking is still possible, though.) On the bad side, it uses more memory, requires copying data around (which may be a problem if shared data is large) and may lead to complicated and hard to understand algorithms.

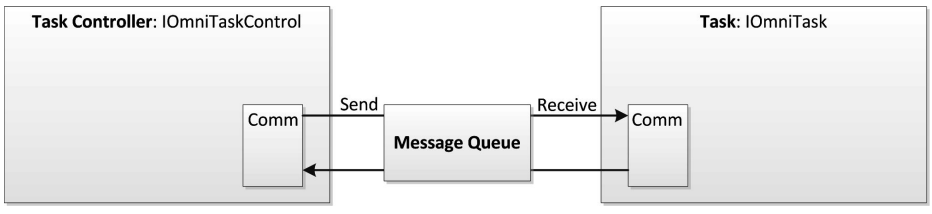
OmniThreadLibrary uses custom lock-free structures to transfer data between the task and its owner (or directly between two tasks). The system is tuned for high data rates and can transfer more than million messages per second. However, in some situations shared data approach is necessary and that's why OmniThreadLibrary adds significant support for synchronisation.

Some would disagree with the OmniThreadLibrary communication structures being called lock-free. In reality, there are no communication mechanisms that would correctly work in a multithreaded world without using locking. The name lock-free only implies that no operating system locking primitives are being used. Instead, OmniThreadLibrary achieves the thread-safeness by using bus locking, a special processor instruction prefix that achieves atomic operation in a multiprocessor system. Bus-locked operations are slower than the normal assembler code, especially as they may stop other cores for the time of the operation execution, but then they are also faster than the operating system locking.

Lock-free (or microlocked) structures in OmniThreadLibrary encompass:

- bounded (size-limited) stack
- bounded (size-limited) queue
- message queue
- dynamic (growing) queue
- blocking collection

OmniThreadLibrary automatically inserts two bounded queues between the task owner (`IOmniTaskControl`) and the task (`IOmniTask`) so that the messages can flow in both directions.



2.7 Message loop required

Due to implementation details, OmniThreadLibrary requires that each thread owner maintains and processes a message queue. This condition is automatically satisfied in VCL and service applications, but running OmniThreadLibrary threads from a console application requires more work. Additional work is also required if you are creating OmniThreadLibrary threads from background threads.

2.7.1 OmniThreadLibrary and console

To correctly use OmniThreadLibrary in a console application, said application must process Windows messages. This is demonstrated in the `62_console` demo which is reproduced below.

```
1  program app_62_console;
2
3  {$APPTYPE CONSOLE}
4
5  uses
6    Windows, Messages, SysUtils,
7    OtlComm, OtlTask, OtlTaskControl, OtlParallel;
8
9  const
10   MSG_STATUS = WM_USER;
11
12   procedure ProcessMessages;
13   var
14     Msg: TMsg;
15   begin
16     while integer(PeekMessage(Msg, 0, 0, 0, PM_REMOVE)) <> 0 do begin
17       TranslateMessage(Msg);
18       DispatchMessage(Msg);
19     end;
20   end;
21
22   function DoTheCalculation(const task: IOmniTask): integer;
23   var
24     i: integer;
25   begin
26     for i := 1 to 5 do begin
27       task.Comm.Send(MSG_STATUS, '... still calculating');
28       Sleep(1000);
29     end;
30     Result := 42;
31   end;
32
33 var
34   calc: IOmniFuture<integer>;
35
36 begin
37   try
38     calc := Parallel.Future<integer>(DoTheCalculation,
39       parallel.TaskConfig.OnMessage(MSG_STATUS,
40         procedure(const task: IOmniTaskControl; const msg: TOmniMessage)
41         begin
42           Writeln(msg.MsgData.AsString);
43         end));
44
45     Writeln('Background thread is calculating ...');
46     while not calc.IsDone do
47       ProcessMessages;
48     Writeln('And the answer is: ', calc.Value);
49
50     if DebugHook <> 0 then
51       Readln;
52   except
53     on E: Exception do
54       Writeln(E.ClassName, ': ', E.Message);
55   end;
56 end.
```

The main program creates a Future which runs a function `DoTheCalculation` and then waits for it to return a value (`while not calc.IsDone`).

The future waits five seconds and each second sends a message back to the owner (`task.Comm.Send(MSG_STATUS, '... still calculating')`). Main thread processes these messages in the `MSG_STATUS` handler.

If you run the program, you'll see that the message "`... still calculating`" is displayed five times with one second delay between two messages. After that, "`And the answer is: 42`" is displayed.

The critical part of this program are two lines:

```
1 while not calc.IsDone do
2   ProcessMessages;
```

If you comment them out, the program will write "`Background thread is calculating ...`", then nothing will happen (visibly) for five seconds and then all messages will be displayed at once.

In this example it is not really critical to process messages. The program would continue to function correctly even when message processing is removed. In other cases, however, all kinds of weird behaviour can occur if messages are not processed. OmniThreadLibrary occasionally uses messages for internal purpose and if you prevent processing of these messages, applications may misbehave. The best approach is to always include periodic calls to ProcessMessages in a console application.

2.7.2 OmniThreadLibrary task started from another task

Similar considerations take order when an OmniThreadLibrary task is started from another OmniThreadLibrary task. The intermediate task (the task which starts another task) must process messages. The easiest way to achieve that is by using the `MsgWait` qualifier when creating a task.

In the `66_ThreadsInThreads` demo, the click on the **OTL from a OTL task** button creates a task that will create a Future.

```
1   FOwnerTask := CreateTask(TWorker.Create(), 'OTL owner')
2     .OnMessage(Self)
3     .OnTerminated(TaskTerminated)
4     .MsgWait // critical, this allows OTL task to process messages
5     .Run;
```

The critical part demonstrated is the call to `MsgWait` which causes internal loop in the task to process Windows messages. Without this `MsgWait` the program would stop working.

The worker does all the work in its `Initialization` method.

```
1  function TWorker.Initialize: boolean;
2  begin
3    Result := inherited Initialize;
4    if Result then begin
5      Task.Comm.Send(WM_LOG, Format('[%d] Starting a Future', [GetCurrentThreadID]));
6      FCalc := Parallel.Future<integer>(Asy_DoTheCalculation,
7        Parallel.TaskConfig
8          .OnMessage(MSG_STATUS,
9            procedure(const workerTask: IOmniTaskControl; const msg: TOmniMessage)
10           begin
11             // workerTask = task controller for Parallel.Future worker thread
12             // Task = TWorker.Task = interface of the TWorker task
13             Task.Comm.Send(WM_LOG, Format('[%d] Future sent a message: %s',
```

```
14                [GetCurrentThreadID, msg.MsgData.AsString]));
15            end)
16          .OnTerminated(
17            procedure
18            begin
19              Task.Comm.Send(WM_LOG, Format('[%d] Future terminated, result = %d',
20                [GetCurrentThreadID, FCalc.Value]));
21              FCalc := nil;
22              Task.Comm.Send(WM_LOG, Format('[%d] Terminating worker',
23                [GetCurrentThreadID]));
24              // Terminate TWorker
25              Task.Terminate;
26            end));
27    end;
28  end;
```

This code executes in a background worker thread. It may look complicated, however, the code simply creates a Future calculation (`FCalc := Parallel.Future<integer>`) and sets up event handlers that will process messages sent from the future (`.OnMessage`) and handle the completion of the future calculation (`.OnTerminated`).

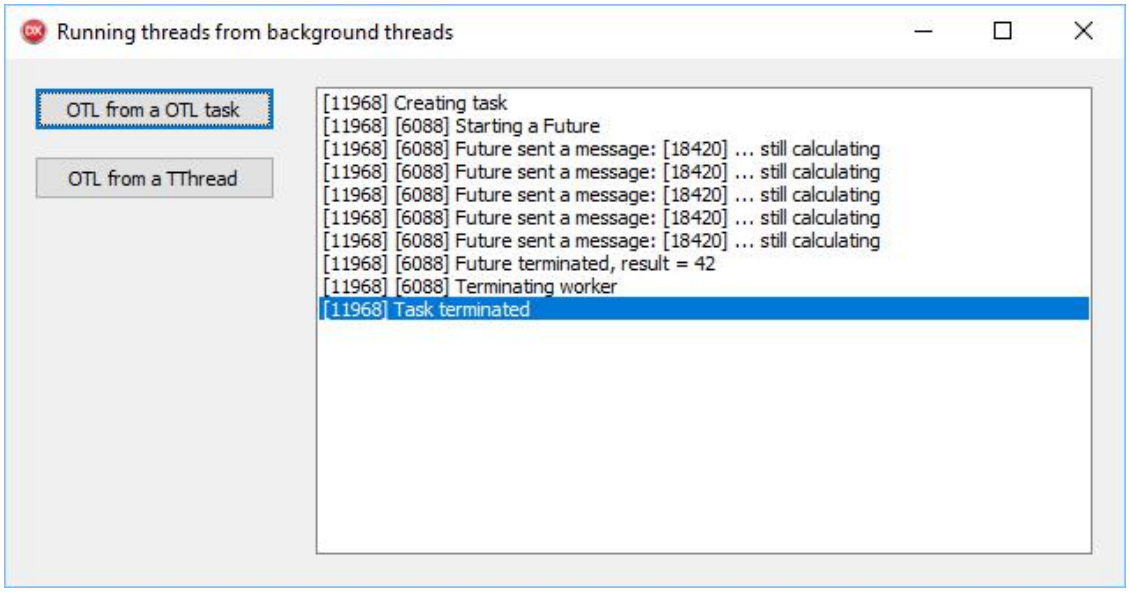
`OnMessage` just takes message that was sent from the future, adds some text and current thread ID and forwards message to the form where it is logged in the `WMLog` method (not shown here).

`OnTerminated` also logs the event, clears the future interface and terminates self (`Task.Terminate`). After that, form's `TaskTerminated` method (not shown here) is called and cleans the task controller interface.

The future itself does nothing special, it just sends five messages with one second delay between them and then returns a value.

```
1  function TWorker.Asy_DoTheCalculation(const task: IOmniTask): integer;
2  var
3    i: integer;
4  begin
5    for i := 1 to 5 do begin
6      task.Comm.Send(MSG_STATUS, Format('[%d] ... still calculating',
7        [GetCurrentThreadID]));
8      Sleep(1000);
9    end;
10   Result := 42;
11 end;
```

When you run the program and click on the button, following text will be displayed (thread IDs – numbers in brackets – will be different in your case, of course).



We can see that Future messages were generated in thread 18420, then passed through the parent thread 6088 and ended in the main thread 11968.

2.7.3 OmniThreadLibrary task started from a TThread

Enhancing a basic Delphi `TThread` is easy with the OmniThreadLibrary and takes only a few simple steps. We have to make sure that any thread messages are periodically processed by calling the `DSiProcessThreadMessages` function from the DSiWin32 unit (or a similar code that calls `PeekMessage` / `TranslateMessage` / `DispatchMessage`).

The `66_ThreadsInThreads` demo contains an example.

```
FThread := TWorkerThread.Create(true);
FThread.OnTerminate := ThreadTerminated;
FThread.FreeOnTerminate := true;
FThread.Start;
```

The main thread method firstly creates a Future and sets up an `.OnMessage` handler which just resends messages to the main thread.

```
procedure TWorkerThread.Execute;
var
  awaited: DWORD;
  calc   : IOmniFuture<integer>;
  handles: array [0..0] of THandle;
begin
  Log('Starting a Future');

  calc := Parallel.Future<integer>(Asy_DoTheCalculation,
    Parallel.TaskConfig
      .OnMessage(MSG_STATUS,
        procedure(const workerTask: IOmniTaskControl; const msg: TOmniMessage)
        begin
          Log('Future sent a message: ' + msg.MsgData.AsString);
        end));

  repeat
    awaited := MsgWaitForMultipleObjects(0, handles, false, INFINITE, QS_ALLPOSTMESSAGE);
    if awaited = WAIT_OBJECT_0 + 0 {handle count} then
      DSiProcessThreadMessages;
  until calc.IsDone;

  Log('Future terminated, result = ' + IntToStr(calc.Value));
  calc := nil;
```

```
25    Log('Terminating worker');
26 end;
```

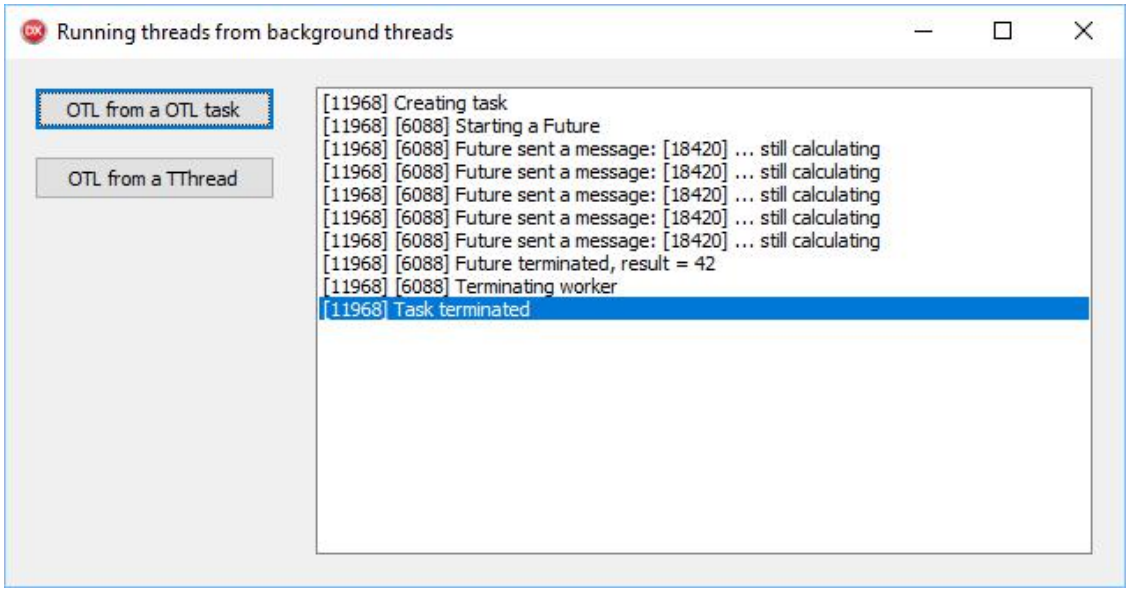
Then it enters a `repeat .. until` loop in which it waits for a Windows message (`MsgWaitForMultipleObjects`), processes all waiting messages (`DSiProcessThreadMessages`) and checks whether the calculation has completed (`calc.IsDone`).

At the end it cleans up the future and exits. That destroys the `TWorkerThread` thread.

Calling `MsgWaitForMultipleObjects`[2] is not strictly necessary. You could just call `DSiProcessThreadMessages` from time to time. It does, however, improve the performance as the code uses no CPU time in such wait.

If you are already using some other kind of wait-and-dispach mechanism in your thread (`WaitForSingleObject`, `WaitForSingleObjectEx`, `WaitForMultipleObjects`, `WaitForMultipleObjectsEx`) then they are easy to convert to `MsgWaitForMultipleObjects` or `MsgWaitForMultipleObjectsEx`.

Running the program shows similar behaviour as in the previous example.



2.8 TOmniValue

A `TOmniValue` (part of the OtlCommon unit) is data type which is central to the whole OmniThreadLibrary. It is used in all parts of the code (for example in a communication subsystem) when type of the data that is to be stored/passed around is not known in advance.

It is implemeted as a smart record (a record with functions and operators) which functions similary to a `Variant` or `TValue` but is faster[3]. It can store following data types:

- simple values (byte, integer, char, double, ...)
- strings (Ansi, Unicode)
- Variant
- objects
- interfaces
- records (in D2009 and newer)

In all cases ownership of reference-counted data types (strings, interfaces) is managed correctly so no memory leaks can occur when such type is stored in a `TOmniValue` variable.

Two kinds of floating-point numbers can be stored in a `TOmniValue` – `double` and `extended`. Former are stored directly in the `TOmniValue` record while latter are wrapped in an `TOmniExtendedData` record, which increases memory usage and decreases performance. The use of `double` floating-point numbers is therefore recommended.

The `TOmniValue` type is too large to be shown in one piece so I'll show various parts of its interface throughout this chapter.

2.8.1 Data access

The content of a `TOmniValue` record can be accessed in many ways, the simplest (and in most cases the most useful) being through the `AsXXX` properties.

```
property AsAnsiString: AnsiString;
property AsBoolean: boolean;
property AsCardinal: cardinal;
property AsDouble: Double;
property AsDateTime: TDateTime;
property AsException: Exception;
property AsExtended: Extended;
property AsInt64: int64 read;
property AsInteger: integer;
property AsInterface: IInterface;
property AsObject: TObject;
property AsOwnedObject: TObject;
property AsPointer: pointer;
property AsString: string;
property AsVariant: Variant;
property AsWideString: WideString;
```

Exceptions can be stored through the `AsObject` property, but there's also a special support for `Exception` data type with its own data access property `AsException`. It is extensively used in the Pipeline abstraction.

While the setters for those properties are pretty straightforward, getters all have a special logic built in which tries to convert data from any reasonable source type to the requested type. If that cannot be done, an exception is raised.

For example, getter for the `AsString` property is called `CastToString` and in turn calls `TryCastToString`, which is in turn a public function of `TOmniValue`.

```
function TOmniValue.CastToString: string;
begin
  if not TryCastToString(Result) then
    raise Exception.Create('TOmniValue cannot be converted to string');
end;

function TOmniValue.TryCastToString(var value: string): boolean;
begin
  Result := true;
  case ovType of
    ovtNull:       value := '';
    ovtBoolean:    value := BoolToStr(AsBoolean, true);
    ovtInteger:    value := IntToStr(ovData);
    ovtDouble,
    ovtDateTime,
    ovtExtended:   value := FloatToStr(AsExtended);
    ovtAnsiString: value := string((ovIntf as IOmniAnsiStringData).Value);
```

```
    ovtString:     value := (ovIntf as IOmniStringData).Value;
    ovtWideString: value := (ovIntf as IOmniWideStringData).Value;
    ovtVariant:    value := string(AsVariant);
    else Result := false;
  end;
end;
```

When you don't know the data type stored in a `TOmniValue` variable and you don't want to raise an exception if compatible data is not available, you can use the `TryCastToXXX` family of functions directly.

```
function  TryCastToAnsiString(var value: AnsiString): boolean;
function  TryCastToBoolean(var value: boolean): boolean;
function  TryCastToCardinal(var value: cardinal): boolean;
function  TryCastToDouble(var value: Double): boolean;
function  TryCastToDateTime(var value: TDateTime): boolean;
function  TryCastToException(var value: Exception): boolean;
function  TryCastToExtended(var value: Extended): boolean;
function  TryCastToInt64(var value: int64): boolean;
function  TryCastToInteger(var value: integer): boolean;
function  TryCastToInterface(var value: IInterface): boolean;
function  TryCastToObject(var value: TObject): boolean;
function  TryCastToPointer(var value: pointer): boolean;
function  TryCastToString(var value: string): boolean;
function  TryCastToVariant(var value: Variant): boolean;
function  TryCastToWideString(var value: WideString): boolean;
```

Alternatively, you can use `CastToXXXDef` functions which return a default value if current value of the `TOmniValue` cannot be converted into required data type.

```
function  CastToAnsiStringDef(const defValue: AnsiString): AnsiString;
function  CastToBooleanDef(defValue: boolean): boolean;
function  CastToCardinalDef(defValue: cardinal): cardinal;
function  CastToDoubleDef(defValue: Double): Double;
function  CastToDateTimeDef(defValue: TDateTime): TDateTime;
function  CastToExceptionDef(defValue: Exception): Exception;
function  CastToExtendedDef(defValue: Extended): Extended;
function  CastToInt64Def(defValue: int64): int64;
function  CastToIntegerDef(defValue: integer): integer;
function  CastToInterfaceDef(const defValue: IInterface): IInterface;
function  CastToObjectDef(defValue: TObject): TObject;
function  CastToPointerDef(defValue: pointer): pointer;
function  CastToStringDef(const defValue: string): string;
function  CastToVariantDef(defValue: Variant): Variant;
function  CastToWideStringDef(defValue: WideString): WideString;
```

They are all implemented in the same manner, similar to the `CastToObjectDef` below.

```
function TOmniValue.CastToObjectDef(defValue: TObject): TObject;
begin
  if not TryCastToObject(Result) then
    Result := defValue;
end;
```

2.8.2 Type testing

For situations where you would like to determine the type of data stored inside the `TOmniValue`, there is the `IsXXX` family of functions.

```
function  IsAnsiString: boolean;
function  IsArray: boolean;
function  IsBoolean: boolean;
function  IsEmpty: boolean;
function  IsException: boolean;
function  IsFloating: boolean;
```

```
function  IsDateTime: boolean;
function  IsInteger: boolean;
function  IsInterface: boolean;
function  IsInterfacedType: boolean;
function  IsObject: boolean;
function  IsOwnedObject: boolean;
function  IsPointer: boolean;
function  IsRecord: boolean;
function  IsString: boolean;
function  IsVariant: boolean;
function  IsWideString: boolean;
```

Alternatively, you can use the `DataType` property.

```
type
  TOmniValueDataType = (ovtNull, ovtBoolean, ovtInteger, ovtDouble, ovtObject,
    ovtPointer, ovtDateTime, ovtException, ovtExtended, ovtString, ovtInterface,
    ovtVariant, ovtWideString, ovtArray, ovtRecord, ovtAnsiString, ovtOwnedObject);

property DataType: TOmniValueDataType;
```

2.8.3 Clearing the content

There are two ways to clear a `TOmniValue` – you can either call its `Clear` method, or you can assign to it a `TOmniValue.Null`.

```
procedure Clear;
class function Null: TOmniValue; static;
```

An example:

```
var
  ov: TOmniValue;

ov.Clear;
// or
ov := TOmniValue.Null;
```

Calling `Clear` is slightly faster.

2.8.4 Operators

`TOmniValue` also implements several `Implicit` operators which help with automatic conversion to and from different data types. Internally, they are implemented as assignment to/from the `AsXXX` property.

```
class operator Equal(const a: TOmniValue; i: integer): boolean;
class operator Equal(const a: TOmniValue; const s: string): boolean;
class operator Implicit(const a: AnsiString): TOmniValue;
class operator Implicit(const a: boolean): TOmniValue;
class operator Implicit(const a: Double): TOmniValue;
class operator Implicit(const a: Extended): TOmniValue;
class operator Implicit(const a: integer): TOmniValue;
class operator Implicit(const a: int64): TOmniValue;
class operator Implicit(const a: pointer): TOmniValue;
class operator Implicit(const a: string): TOmniValue;
class operator Implicit(const a: IInterface): TOmniValue;
class operator Implicit(const a: TObject): TOmniValue;
class operator Implicit(const a: Exception): TOmniValue;
class operator Implicit(const a: TOmniValue): AnsiString;
class operator Implicit(const a: TOmniValue): int64;
class operator Implicit(const a: TOmniValue): TObject;
class operator Implicit(const a: TOmniValue): Double;
class operator Implicit(const a: TOmniValue): Exception;
class operator Implicit(const a: TOmniValue): Extended;
```

```
20 class operator Implicit(const a: TOmniValue): string;
21 class operator Implicit(const a: TOmniValue): integer;
22 class operator Implicit(const a: TOmniValue): pointer;
23 class operator Implicit(const a: TOmniValue): WideString;
24 class operator Implicit(const a: TOmniValue): boolean;
25 class operator Implicit(const a: TOmniValue): IInterface;
26 class operator Implicit(const a: WideString): TOmniValue;
27 class operator Implicit(const a: Variant): TOmniValue;
28 class operator Implicit(const a: TDateTime): TOmniValue;
29 class operator Implicit(const a: TOmniValue): TDateTime;
```

Implicit conversion to/from `TDateTime` is supported only in Delphi XE and newer.

Two `Equal` operators simplify comparing `TOmniValue` to an `integer` and `string` data.

2.8.5 Using with generic types

Few methods simplify using `TOmniValue` with class and record data.

```
1 class function CastFrom<T>(const value: T): TOmniValue; static;
2 function  CastTo<T>: T;
3 function  CastToObject<T: class>: T; overload;
4 function  ToObject<T: class>: T;
5 class function Wrap<T>(const value: T): TOmniValue; static;
6 function  Unwrap<T>: T;
```

`CastFrom<T>` converts any type into a `TOmniValue`. In Delphi 2009, this function is severely limited as only very simple types (integer, object) are supported. Starting with Delphi 2010, `TValue` type is used to facilitate the conversion and all data types supported by the `TOmniValue` can be converted.

`CastTo<T>` converts `TOmniValue` into any other type. In Delphi 2009 same limitations apply as for `CastFrom<T>`.

`CastToObject<T>` (available in Delphi 2010 and newer) performs a hard cast with no type checking. It is equivalent to using `T(omnivalue.AsObject)`

`ToObject<T>` (available in Delphi 2010 and newer) casts the object to type `T` with type checking. It is equivalent to using `omnivalue.AsObject as T`.

`Wrap<T>` [3.06] wraps any data type in an instance of `TOmniRecordWrapper<T>` and stores this value in a `TOmniValue` variable.

`Unwrap<T>` [3.06] unwraps a `TOmniValue` holding a `TOmniRecordWrapper<T>` and returns owned value of type `T`. It has to be called in this form: `omnivalue.Unwrap<T>()`. Trailing `()` is required.

`Wrap` and `Unwrap` are especially useful as they allow you to store `TMethod` data (event handlers) in a `TOmniValue` variable.

2.8.6 Array access

Each `TOmniValue` can contain an array of other `TOmniValue`s. Internally, they are stored in a `TOmniValueContainer` object. This object can be accessed directly by reading the `AsArray` property.

```
1 property AsArray: TOmniValueContainer read GetAsArray;
```

`IsArray` can be used to test whether a `TOmniValue` contains an array of values.

Arrays can be accessed by an integer indexes (starting with 0), or by string indexes (named access). Integer-indexed arrays are created by calling `TOmniValue.Create` and string-indexed arrays are created by calling `TOmniValue.CreateNamed`.

```
constructor Create(const values: array of const);
constructor CreateNamed(const values: array of const);
```

In the latter case, elements of the `values` parameter must alternate between names (string indexes) and values.

```
ov := TOmniValue.CreateNamed(
        ['Key1', 'Value of ov[""Key1""]',
         'Key2', 'Value of ov[""Key2""]'
        ]);
```

In the example above, both `ov[0]` and `ov['Key1']` would return the same string, namely `'Value of ov[''Key1'']'`.

Array elements can be accessed with the `AsArrayItem` property, by using an integer index (for integer-indexed arrays), a string index (for string-indexed arrays), or a `TOmniValue` index. In the last case, the type of data stored inside the `TOmniValue` index parameter will determine how the array element is accessed. This last form is not available in Delphi 2007, where `AsArrayItemOV` should be used instead.

All forms of `AsArrayItem` allow extending an array. If you write data into an index which doesn't already exist, the array will automatically grow to accomodate the new value.

```
property AsArrayItem[idx: integer]: TOmniValue; default;
property AsArrayItem[const name: string]: TOmniValue; default;
property AsArrayItem[const param: TOmniValue]: TOmniValue; default;
property AsArrayItemOV[const param: TOmniValue]: TOmniValue;
```

If you want to test whether an array element exists, use the `HasArrayItem` function.

```
function  HasArrayItem(idx: integer): boolean; overload;
function  HasArrayItem(const name: string): boolean; overload;
function  HasArrayItem(const param: TOmniValue): boolean; overload;
```

Starting with Delphi 2010 `TOmniValue` also implements functions for converting data to and from `TArray<T>` for any supported type. `CastFrom<T>` and `CastTo<T>` functions are used internally to do the conversion.

```
class function FromArray<T>(const values: TArray<T>): TOmniValue; static;
function  ToArray<T>: TArray<T>;
```

2.8.7 Handling records

A record `T` can be stored inside a `TOmniValue` by calling the `FromRecord<T>` function. To extract the data back into a record, use the `ToRecord<T>` function.

```
class function FromRecord<T: record>(const value: T): TOmniValue; static;
function  ToRecord<T>: T;
```

An example:

```
var
  ts: TTimeStamp;
  ov: TOmniValue;

```

```
5 ov := TOmniValue.FromRecord<TTimeStamp>(ts);
6 ts := ov.ToRecord<TTimeStamp>;
```

`TOmniValue` jumps through quite some hoops to store a record. It is first converted into a `TOmniRecordWrapper` which is then wrapped inside an `IOmniAutoDestroyObject` interface to provide a reference-counted lifetime management.

Because of that convoluted process, storing records inside `TOmniValue` is not that fast.

```
1 class function TOmniValue.FromRecord<T>(const value: T): TOmniValue;
2 begin
3   Result.SetAsRecord(
4     CreateAutoDestroyObject(
5       TOmniRecordWrapper<T>.Create(value)));
6 end;
```

2.8.8 Object ownership

`TOmniValue` can take an ownership of a `TObject`-type data. To achieve that, you can either assign an object to the `AsOwnedObject` property or set the `OwnsObject` property to `True`.

```
1 property  AsOwnedObject: TObject;
2 function  IsOwnedObject: boolean;
3 property OwnsObject: boolean;
```

When a object-owning `TOmniValue` goes out of scope, the owned object is automatically destroyed.

You can change the ownership status at any time by setting the `OwnsObject` property.

2.8.9 Working with TValue

Starting with Delphi 2010 `TOmniValue` provides an `AsTValue` property and corresponding `Implicit` operator so you can easily convert a `TValue` data into a `TOmniValue` and back.

```
1 class operator Implicit(const a: TValue): TOmniValue; inline;
2 class operator Implicit(const a: TOmniValue): TValue; inline;
3 property AsTValue: TValue;
```

2.8.10 Low-level methods

For programmers with special requirements (and for internal OmniThreadLibrary use), `TOmniValue` exposes following public methods.

```
1 procedure _AddRef;
2 procedure _Release;
3 procedure _ReleaseAndClear;
4 function  RawData: PInt64;
5 procedure RawZero;
```

`_AddRef` increments reference counter of stored data if `TOmniValue` contains such data.

`_Release` decrements reference counter of stored data if `TOmniValue` contains such data.

`_ReleaseAndClear` is just a shorthand for calling a `_Release` followed by a call to `RawZero`.

`RawData` returns pointer to the data stored in the `TOmniValue`.

`RawZero` clears the stored data without decrementing the reference counter.

2.9 TOmniValueObj

The OtlCommon unit implements a simple object which can wrap a `TOmniValue` for situations where you would like to store it inside a data structure that only supports object types.

```
TOmniValueObj = class
  constructor Create(const value: TOmniValue);
  property Value: TOmniValue read FValue;
end;
```

2.10 Fluent interfaces

OmniThreadLibrary heavily uses fluent interface approach. Most of functions in OmniThreadLibrary interfaces are returning `Self` as the result. Take for example this declaration of the Pipeline abstraction, slightly edited for brevity.

```
IOmniPipeline = interface
  procedure Cancel;
  function  From(const queue: IOmniBlockingCollection): IOmniPipeline;
  function  HandleExceptions: IOmniPipeline;
  function  NumTasks(numTasks: integer): IOmniPipeline;
  function  OnStop(const stopCode: TProc): IOmniPipeline;
  function  Run: IOmniPipeline;
  function  Stage(
    pipelineStage: TPipelineSimpleStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stage(
    pipelineStage: TPipelineStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stage(
    pipelineStage: TPipelineStageDelegateEx;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineSimpleStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineStageDelegateEx;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Throttle(numEntries: integer; unblockAtCount: integer = 0):
    IOmniPipeline;
  function  WaitFor(timeout_ms: cardinal): boolean;
end;
```

As you can see, most of the functions return the `IOmniPipeline` interface. In code, this is implemented by returning `Self`.

```
function TOmniPipeline.From(
  const queue: IOmniBlockingCollection): IOmniPipeline;
begin
  opInput := queue;
  Result := Self;
end;
```

This allows calls to such interfaces to be chained. For example, the following code from the Pipeline section of the book shows how to use `Parallel.Pipeline` without ever storing the resulting interface in a variable.

```
var
  sum: integer;

sum := Parallel.Pipeline
```

```
  .Stage(
    procedure (const input, output: IOmniBlockingCollection)
    var
      i: integer;
    begin
      for i := 1 to 1000000 do
        output.Add(i);
    end)
  .Stage(
    procedure (const input: TOmniValue; var output: TOmniValue)
    begin
      output := input.AsInteger * 3;
    end)
  .Stage(
    procedure (const input, output: IOmniBlockingCollection)
    var
      sum: integer;
      value: TOmniValue;
    begin
      sum := 0;
      for value in input do
        Inc(sum, value);
      output.Add(sum);
    end)
  .Run.Output.Next;
```

If you don’t like fluent interface approach, don’t worry. OmniThreadLibrary can be used without it. You can always call a function as if it is a procedure and compiler will just throw away the result.

The example above could be rewritten as such.

```
var
  sum: integer;
  pipe: IOmniPipeline;

pipe := Parallel.Pipeline;
pipe.Stage(
  procedure (const input, output: IOmniBlockingCollection)
  var
    i: integer;
  begin
    for i := 1 to 1000000 do
      output.Add(i);
  end);
pipe.Stage(
  procedure (const input: TOmniValue; var output: TOmniValue)
  begin
    output := input.AsInteger * 3;
  end);
pipe.Stage(
  procedure (const input, output: IOmniBlockingCollection)
  var
    sum: integer;
    value: TOmniValue;
  begin
    sum := 0;
    for value in input do
      Inc(sum, value);
    output.Add(sum);
  end);
pipe.Run;
sum := pipe.Output.Next;
```

3. High-level multithreading

Face it – multithreading programming is hard. It is hard to design a multithread program, it is hard to write and test it and it is insanely hard to debug it. To alleviate this problem, OmniThreadLibrary introduces a number of pre-packaged multithreading solutions; so-called abstractions.

The idea behind the high-level abstractions is that the user should just choose appropriate abstraction and write the worker code, while the OmniThreadLibrary provides the framework that implements the tricky multithreaded parts, takes care of synchronisation and so on.

3.1 Introduction

High-level abstractions are implemented in the OtlParallel unit. They are all created through the factory class `Parallel`. High-level code intensively uses anonymous methods

and generics which makes Delphi 2009 the minimum supported version. As the implementation of generics in D2009 and D2010 is not very stable, I' d recommend using at least Delphi XE.

3.1.1 A lifecycle of an abstraction

Typical factory from the `Parallel` class returns an interface. For example, `Parallel` implements five `Join` overloads which create a Join abstraction.

```
1 class function  Join: IOmniParallelJoin; overload;
2 class function  Join(const task1, task2: TProc):
3   IOmniParallelJoin; overload;
4 class function  Join(const task1, task2: TOmniJoinDelegate):
5   IOmniParallelJoin; overload;
6 class function  Join(const tasks: array of TProc):
7   IOmniParallelJoin; overload;
8 class function  Join(const tasks: array of TOmniJoinDelegate):
9   IOmniParallelJoin; overload;
```

Important fact to keep in mind is that the Join abstraction (or any other abstraction returned from any of the `Parallel` factories) will only be alive as long as this interface is not destroyed. For example, the following code fragment will function fine.

```
1 procedure Test_OK;
2 begin
3   Parallel.Join(
4     procedure
5     begin
6       Sleep(10000);
7     end,
8     procedure
9     begin
10      Sleep(15000);
11    end;
12  end
13  ).Execute;
14 end;
```

The interface returned from the `Parallel.Join` call is stored in a hidden variable, which will only be destroyed while executing the `end` statement of the `Test_OK` procedure. As the `Execute` waits for all subtasks to complete, `Join` will complete its execution before the `end` is executed and before the interface is destroyed.

Modify the code slightly and it will not work anymore.

```
procedure Test_Fail;
begin
  Parallel.Join(
    procedure
    begin
      Sleep(10000);
    end,
    procedure
    begin
      Sleep(15000);
    end;
  end
  ).NoWait.Execute;
end;
```

Because of the `NoWait` modifier, `Execute` will not wait for subtasks to complete but will return immediately. Next, the code behind the `end` will be executed and will destroy the abstraction while the subtasks are still running.

In such cases, it is important to keep the interface in a global field (typically it will be stored inside a form or another object) which will stay alive until the abstraction has stopped.

```
procedure Test_OK_Again;
begin
  FJoin := Parallel.Join(
    procedure
    begin
      Sleep(10000);
    end,
    procedure
    begin
      Sleep(15000);
    end;
  end
  ).NoWait.Execute;
end;
```

This leads to a new problem – when should this interface be destroyed? The answer depends on the abstraction used. Some abstractions provide `OnStop` method which can be used for this purpose. For other abstractions, you should use termination handler of the task configuration block.

3.1.2 Anonymous methods, procedures, and methods

High-level abstractions are very much based on anonymous methods. They are used all over the OtlParallel unit and they will also be used in your own code as they provide the simplest way to interact with the high-level threading. All of delegates (pieces of code that you ‘plug in’ into the high-level infrastructure) are declared as anonymous methods.

That, however, does not force you to write anonymous methods to use high-level multithreading. Thanks to the Delphi compiler, you can always provide a normal function/procedure or a method when an anonymous method is required.

For example, let’s take a look at the `IOmniWorkItemConfig` interface. It defines (along with other stuff) method `OnExecute` which accepts a delegate of type `TOmniBackgroundWorkerDelegate`.

```
TOmniBackgroundWorkerDelegate = reference to procedure (
  const workItem: IOmniWorkItem);

IOmniWorkItemConfig = interface
  function  OnExecute(const aTask: TOmniBackgroundWorkerDelegate):
```

```
6     IOmniWorkItemConfig;
7   ...
8 end;
```

Let's assume a variable of the appropriate type, `config: IOmniWorkItemConfig`. You can then call the `OnExecute` method using an anonymous method.

```
1 config.OnExecute(
2   procedure(const workItem: IOmniWorkItem)
3   begin
4     ...
5   end);
```

Alternatively, you could declare a 'normal' procedure with the same signature (with the same parameters) and pass it to the `OnExecute` call.

```
1 procedure OnConfigExecute(const workItem: IOmniWorkItem);
2 begin
3   ...
4 end;
5
6 config.OnExecute(OnConfigExecute);
```

The third option is to pass a method of some class to the `OnExecute`.

```
1 procedure TMyClass.OnConfigExecute(const workItem: IOmniWorkItem);
2 begin
3   ...
4 end;
5
6 procedure TMyClass.DoConfig;
7 begin
8   config.OnExecute(OnConfigExecute);
9 end;
```

These three options are valid whenever an anonymous method delegate can be used.

3.1.3 Pooling

Starting a thread is relatively slow operation, which is why threads, used in the high-level abstractions are not constantly created and destroyed. Rather than that, they are allocated from a thread pool.

All high-level abstractions are using the same thread pool, `GlobalParallelPool`. It has public visibility – just in case you have to configure its parameters.

```
1 function GlobalParallelPool: IOmniThreadPool;
```

This behaviour can be overridden by using the task configuration block.

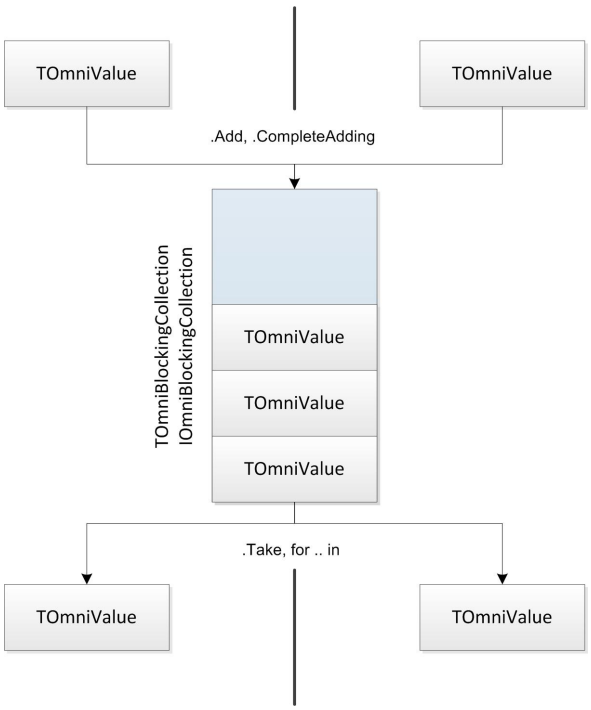
The following rules hold for all tasks created by the high-level abstractions:

- If a task configuration block is not specified, task is created in the `OtlParallel.GlobalParallelPool` pool.
- If a task configuration block is specified:
  - If `ThreadPool(aPool)` is called, task is created in the `aPool` pool.
  - If `ThreadPool(nil)` is called, task is created in the default `OtlThreadPool.GlobalOmniThreadPool` pool.
  - If `NoThreadPool` is called, task is not created in a pool but instead uses a newly created thread (IOW, a `Run` is used instead of a `Schedule`).
  - Otherwise, task is created in the `OtlParallel.GlobalParallelPool` pool (same as when a

task configuration block is not specified).

3.2 Blocking collection

The blocking collection is a Delphi clone of .NET 4 BlockingCollection. It is a thread-safe collection that provides multiple simultaneous readers and writers. This implementation of blocking collection works only with the TOmniValue elements, which is not a big limitation provided that TOmniValue can store anything, including a class, an interface and a record.



See also demo 33_BlockingCollection.

3.2.1 IOmniBlockingCollection

The blocking collecting is exposed as an interface that lives in the OtlCollections unit.

```
IOmniBlockingCollection = interface
  procedure Add(const value: TOmniValue);
  procedure CompleteAdding;
  function  GetEnumerator: IOmniValueEnumerator;
  function  IsCompleted: boolean;
  function  IsEmpty: boolean;
  function  IsFinalized: boolean;
  function  Next: TOmniValue;
  procedure ReraiseExceptions(enable: boolean = true);
  procedure SetThrottling(highWatermark, lowWatermark: integer);
  function  Take(var value: TOmniValue): boolean;
  function  TryAdd(const value: TOmniValue): boolean;
  function  TryTake(var value: TOmniValue;
    timeout_ms: cardinal = 0): boolean;
  property ContainerSubject: TOmniContainerSubject
    read GetContainerSubject;
  property Count: integer read GetApproxCount;
end;
```

There's also a class TOmniBlockingCollection which implements this interface. This class is public and can be used in your code.

TOmniBlockingCollection also implements a class function ToArray<T> which extracts all data from

a blocking collection and stores it into an array.

`ToArray<T>` is only available in Delphi 2010 and newer.

```
1 class function TOmniBlockingCollection.ToArray<T>
2   (coll: IOmniBlockingCollection): TArray<T>;
```

Demo `61_CollectionToArray` shows how to use `ToArray<T>`.

The blocking collection works in the following way:

- `Add` will add new value to the collection (which is internally implemented as a queue).
- `CompleteAdding` tells the collection that all data has been added. From now on, calling `Add` will raise an exception.
- `TryAdd` is the same as `Add` except that it doesn't raise an exception but returns `False` if the value can't be added (if `CompleteAdding` was already called)
- `IsCompleted` returns `True` after the `CompleteAdding` has been called.
- `IsEmpty` returns `True` if collection is empty. [3.07]
- `Count` returns number of elements in the collection. [3.07]
- `IsFinalized` returns `True` if `CompleteAdding` has been called **and** the collection is empty.
- `Next` retrieves next element from the collection (by calling `Take`) and returns it as the result. If `Take` fails, an `ECollectionCompleted` exception is raised.
- `ReraiseExceptions` enables or disables internal exception-checking flag. Initially this flag is disabled. When the flag is enabled and `Take` or `TryTake` retrieve a `TOmniValue` holding an exception object, this exception is immediately raised.
- `SetThrottling` enables the throttling mechanism. See Throttling for more information.
- `Take` reads next value from the collection. If there's no data in the collection, `Take` will block until the next value is available. If, however, any other thread calls `CompleteAdding` while the `Take` is blocked, `Take` will unblock and return `False`.

- `TryTake` is the same as `Take` except that it has a timeout parameter specifying maximum time the call is allowed to wait for the next value. If `INFINITE` is passed in the timeout parameter, `TryTake` will block until the data is available or `CompleteAdding` is called.
- `ContainerSubject` property enables the blocking collection to partake in lock-free collection observing mechanism.

Enumerator calls `Take` in the `MoveNext` method and returns the value returned from `Take`. Enumerator will therefore block when there is no data in the collection. The usual way to stop the enumerator is to call `CompleteAdding` which will unblock all pending `MoveNext` calls and stop enumeration.

3.2.2 Throttling

Normally, a blocking collection can grow without limits and can fill up the available memory. If the algorithm doesn't prevent this intrinsically, it is sometimes useful to set up throttling, a mechanism which blocks additions when the blocking collection size reaches some predetermined value (high watermark) and which allows additions again when the size reaches another predetermined value (low watermark).

```
1 procedure SetThrottling(highWatermark, lowWatermark: integer);
```

The behaviour of `Add`, `TryAdd`, `Take` and `TryTake` is modified if the throttling is used.

When `Add` or `TryAdd` is called and number of elements in the blocking collection equals `highWatermark`, the code blocks. It will only continue if the number of elements in the collection falls below the `lowWatermark` or if the `CompleteAdding` is called.

When `Take` or `TryTake` take an element from the collection and adding is temporarily blocked because of the throttling and new number of elements in the collection is now below the `lowWatermark`, all waiting `Add` and `TryAdd` calls will be unblocked.

3.3 Task configuration

High-level abstractions will do most of the hard work for you, but sometimes you'll still have to apply some configuration to the low-level parallel tasks (entities represented with the `IOmniTask` interface). The mechanism for doing low-level configuration is called task configuration block.

Task configuration block, or `IOmniTaskConfig`, is an interface returned from the `Parallel.TaskConfig` factory function.

```
class function Parallel.TaskConfig: IOmniTaskConfig;
begin
  Result := TOmniTaskConfig.Create;
end;
```

This interface contains various functions that set up messaging handlers, termination handlers and so on. All function return interface itself so they can be used in a fluent manner.

```
IOmniTaskConfig = interface
  procedure Apply(const task: IOmniTaskControl);
  function  CancelWith(
    const token: IOmniCancellationToken): IOmniTaskConfig;
  function  MonitorWith(
    const monitor: IOmniTaskControlMonitor): IOmniTaskConfig;

  function  NoThreadPool: IOmniTaskConfig;
  function  OnMessage(
    eventDispatcher: TObject): IOmniTaskConfig; overload;
  function  OnMessage(
    eventHandler: TOmniTaskMessageEvent): IOmniTaskConfig; overload;
  function  OnMessage(
    msgID: word;
    eventHandler: TOmniTaskMessageEvent): IOmniTaskConfig; overload;
  function  OnMessage(
    msgID: word;
    eventHandler: TOmniOnMessageFunction): IOmniTaskConfig; overload;
  function  OnTerminated(
    eventHandler: TOmniTaskTerminatedEvent): IOmniTaskConfig; overload;
  function  OnTerminated(
    eventHandler: TOmniOnTerminatedFunction): IOmniTaskConfig; overload;
  function  OnTerminated(
    eventHandler: TOmniOnTerminatedFunctionSimple): IOmniTaskConfig;
    overload;
  function  SetPriority(threadPriority: TOTLThreadPriority): IOmniTaskConfig;
  function  ThreadPool(const threadPool: IOmniThreadPool): IOmniTaskConfig;
  function  WithCounter(
    const counter: IOmniCounter): IOmniTaskConfig;
  function  WithLock(
    const lock: TSynchroObject;
    autoDestroyLock: boolean = true): IOmniTaskConfig; overload;
  function  WithLock(
    const lock: IOmniCriticalSection): IOmniTaskConfig; overload;
end;
```

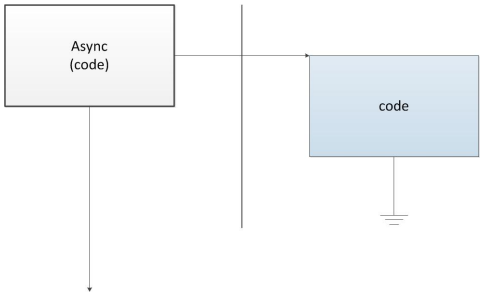
- `Apply` applies task configuration block to a task. It is used internally in the OtlParallel unit.
- `CancelWith` shares cancellation token with the task.
- `MonitorWith` attaches task to the monitor.

- `NoThreadPool` instructs the OtlParallel architecture to create new thread to execute worker task instead of running task in a thread pool. [3.07.2]
- `OnMessage` functions set up message dispatch to an object (for example, a form) or to an event handler. In the latter case, event handler can be set for specific message ID or for all messages.
- `OnTerminated` sets up termination handler which is called when the task terminates.
- `SetPriority` specifies priority for worker tasks. Default priority normal will be used if this method is not called.
- `ThreadPool` specifies the thread pool used for executing worker tasks. Default thread pool `GlobalParallelPool` (singleton, defined in the OtlParallel unit) is used for executing worker tasks if this method is not called. If a `nil` is passsed as an argument, default OmniThreadLibrary pool `OtlThreadPool.GlobalOmniThreadPool` will be used.
- `WithCounter` shares a counter with the task.
- `WithLock` shares a lock object with the task.

Examples of task configuration block usage are demonstrated in demo 47_TaskConfig. Some examples are also given in sections describing individual abstractions.

3.4 Async

Async is the simplest of high-level abstractions and is typically used for fire and forget scenarios. To create an Async task, call Parallel.Async.



When you call `Parallel.Async`, code is started in a new thread (indicated by the bold vertical line) and both main and background threads continue execution. At some time, background task completes execution and disappears.

See also demo `46_Async`.

Example:

```
1 Parallel.Async(
2   procedure
3   begin
4     MessageBeep($FFFFFFFF);
5   end);
```

This simple program creates a background task with a sole purpose to make some noise. The task is coded as an anonymous method but you can also use a normal method or a normal procedure for the task code.

The `Parallel` class defines two `Async` overloads. The first accepts a parameter-less background task and an optional task configuration block and the second accepts a background task with an `IOmniTask` parameter and an optional task configuration block.

```
1 type
2   TOmniTaskDelegate = reference to procedure(const task: IOmniTask);
```

```
3
4   Parallel = class
5     class procedure Async(task: TProc;
6       taskConfig: IOmniTaskConfig = nil); overload;
7     class procedure Async(task: TOmniTaskDelegate;
8       taskConfig: IOmniTaskConfig = nil); overload;
9     ...
10  end;
```

The second form is useful if the background code needs access to the `IOmniTask` interface, for example to send messages to the owner or to execute code in the owner thread (typically that will be the main thread).

The example below uses Async task to fetch the contents of a web page (by calling a mysterious function HttpGet) and then uses `Invoke` to execute a code that logs the length of the result in the main thread.

```
1 Parallel.Async(
2   procedure (const task: IOmniTask)
3   var
4     page: string;
5   begin
6     HttpGet('otl.17slon.com', 80, 'tutorials.htm', page, '');
7     task.Invoke(
8       procedure
9       begin
10        lbLogAsync.Items.Add(Format('Async GET: %d ms; page length = %d',
11          [time, Length(page)]))
12      end);
13  end);
```

The same result could be achieved by sending a message from the background thread to the main thread. In the example below, `TaskConfig` block is used to configure message handler.

```
1 const
2   WM_RESULT = WM_USER;
3
4 procedure LogResult(const task: IOmniTaskControl; const msg: TOmniMessage);
5 begin
6   lbLogAsync.Items.Add(Format('Async GET: %d ms; page length = %d',
7     [time, Length(page)]))
8 end;
9
10 Parallel.Async(
11   procedure (const task: IOmniTask)
12   var
13     page: string;
14   begin
15     HttpGet('otl.17slon.com', 80, 'tutorials.htm', page, '');
16     task.Comm.Send(WM_RESULT, page);
17   end,
18   TaskConfig.OnMessage(WM_RESULT, LogResult)
19 );
```

Let me warn you that in cases where you want to return a result from a background task, Async abstraction is not the most appropriate. You would be better off using a Future.

3.4.1 Handling exceptions

If the background code raises unhandled exception, OmniThreadLibrary catches this exception and re-raises it in the `OnTerminated` handler. This way the exception travels from the background thread to the owner thread where it can be processed.

As the `OnTerminated` handler executes at an unspecified moment when Windows are processing window messages, there is no good way to catch this message with a try..except block. The caller must install its own `OnTerminated` handler instead and handle exception there.

The following example uses `OnTerminated` handler to detach fatal exception from the task, log the exception details and destroy the exception object.

```
1  Parallel.Async(
2    procedure
3    begin
4      Sleep(1000);
5      raise Exception.Create('Exception in Async');
6    end,
7    Parallel.TaskConfig.OnTerminated(
8      procedure (const task: IOmniTaskControl)
9      var
10       excp: Exception;
11     begin
12       if assigned(task.FatalException) then begin
13         excp := task.DetachException;
14         Log('Caught async exception %s:%s',[excp.ClassName, excp.Message]);
15         FreeAndNil(excp);
16       end;
17     end
18   ));
```

If you don't install an `OnTerminated` handler, exception will be handled by the application-level filter, which will by default cause a message box to appear.

See also demo `48_OtlParallelExceptions`.

3.5 Async/Await

Async/Await is a simplified version of the Async abstraction which mimics the .NET Async/Await mechanism.[4]

In short, Async/Await accepts two parameterless anonymous methods. The first one is executed in a background thread and the second one is executed in the main thread after the background thread has completed its work.

See also demo `53_AsyncAwait`.

Using Async/Await you can, for example, create a background operation which is triggered by a click and which re-enables button after the background job has been completed.

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  var
3    button: TButton;
4  begin
5    button := Sender as TButton;
6    button.Caption := 'Working ...';
7    button.Enabled := false;
8    Async(
9      // executed in a background thread
10     procedure begin
11       Sleep(5000);
12     end).
13   Await(
14     // executed in the main thread after
15     // the anonymous method passed to
16     // Async has completed its work
```

```
17      procedure begin
18        button.Enabled := true;
19        button.Caption := 'Done!';
20      end);
21  end;
```

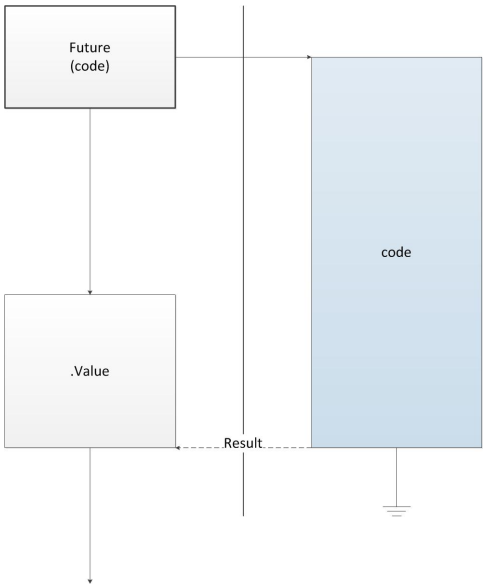
Keep in mind that Async is invoked by calling `Parallel.Async` and Async/Await by calling `Async`.

Exceptions in the `Async` part are currently not handled by the OmniThreadLibrary.

3.6 Future

A Future is a background calculation that returns a result. To create the abstraction, call `Parallel.Future<T>` (where `T` is the type returned from the calculation). This will return a result of type `IOmniFuture<T>`, which is the interface you will use to work with the background task.

To get the result of the calculation, call the `.Value` method on the interface returned from the `Parallel.Future` call.



When you call `Parallel.Future`, a background task is started immediately. The task continues with its (possibly long) execution and the main thread can do some other work. When you need the result of the background calculation, call the `.Value` method, which will return the result immediately if it is ready or which will wait for the background code to complete its work if necessary.

See also demo `39_Future`.

Example:

```
1  var
2    FCalculation: IOmniFuture<integer>;
3
4  procedure StartCalculation;
5  begin
6    FCalculation := Parallel.Future<integer>(
7      function: integer
8      var
9        i: integer;
10     begin
11       Result := 0;
12       for i := 1 to 100000 do
```

```
13        Result := Result + i;
14    end
15  );
16 end;
17
18 function GetResult: integer;
19 begin
20   Result := FCalculation.Value;
21   FCalculation := nil;
22 end;
```

The `Parallel` class implements two `Future<T>` overloads. The first accepts a parameter-less background task and an optional task configuration block and the second accepts a background task with an `IOmniTask` parameter and an optional task configuration block.

```
1  type
2    TOmniFutureDelegate<T> = reference to function: T;
3    TOmniFutureDelegateEx<T> =
4      reference to function(const task: IOmniTask): T;
5
6    Parallel = class
7      class function Future<T>(action: TOmniFutureDelegate<T>;
8        taskConfig: IOmniTaskConfig = nil): IOmniFuture<T>; overload;
9      class function Future<T>(action: TOmniFutureDelegateEx<T>;
10       taskConfig: IOmniTaskConfig = nil): IOmniFuture<T>; overload;
11     ...
12   end;
```

The second form is useful if the background code needs access to the `IOmniTask` interface, for example to send messages to the owner or to execute code in the owner thread (typically that will be the main thread). See the Async section for an example.

A Future task always wraps a function of some type. In the example above the function added numbers from 1 to 100000 together and returned an integer result. That's why the Future task was created by calling `Parallel.Future<integer>` and why the result – the interface that provides a way to manage the task – is declared as `IOmniFuture<integer>`. But a Future could equally well return result of any type – a string or a date/time or even a record, class or interface.

The following example is a rewrite of the Async example. It uses the same mysterious `HttpGet` function but it is wrapped in a more flexible way. Function `StartHttpGet` accepts a url parameter specifying which page to retrieve from the web server. It then creates a future returning a string and passes it a simple code to execute – a one-liner anonymous function that only calls the already known `HttpGet`.

This example illustrates two important points:

- Anonymous methods are great for capturing variables. If you look at the `StartHttpGet` again, you'll see that the url parameter is used from two different threads. `StartHttpGet` is called in the main thread and the anonymous code is executed in the background thread. Still, the parameter somehow makes it across this thread boundary – and that is done by the anonymous method magic.
- Don't write long and complicated anonymous methods – it is better to call external method from the anonymous code and make the long and complicated calculation in that external method (or in code called from that method etc).

```
1 var
2   FGetFuture: IOmniFuture<string>;
3
4 function HttpGet(const url: string): string;
```

```
begin
  // this function fetches a page from the web server
  // and returns its contents
end;

procedure StartHttpGet(const url: string);
begin
  FGetFuture := Parallel.Future<string>(
    function: string
    begin
      Result := HttpGet(url);
    end
  );
end;

function GetResult: string;
begin
  Result := FGetFuture.Value;
  FGetFuture := nil;
end;
```

3.6.1 IOmniFuture<T> interface

The `IOmniFuture<T>` interface implements other methods besides the `Value`.

```
type
  IOmniFuture<T> = interface
    procedure Cancel;
    function  DetachException: Exception;
    function  FatalException: Exception;
    function  IsCancelled: boolean;
    function  IsDone: boolean;
    function  TryValue(timeout_ms: cardinal; var value: T): boolean;
    function  Value: T;
    function  WaitFor(timeout_ms: cardinal): boolean;
   end;
```

The interface implements exception-handling functions, cancellation support and functions that check if the background calculation has completed.

3.6.2 Completion detection

When you call the `Value` function you don’ t know ahead what will happen. If the background code has already calculated the result, the `Value` call will return immediately. Otherwise, the caller thread will be blocked until the result is available and if you are executing a long calculation (or if the web or database connection did not succeed and is now waiting for a timeout to occur) this may last a while. If you created the future in the main thread, then your whole application will be blocked until `Value` returns.

There are few ways around this problem. One is to periodically call the `IsDone` function. It will return `False` while the background calculation is still working and `True` once the result is available. Another option is to call `WaitFor` with some (small) timeout. `WaitFor` will wait specified number of milliseconds and will return `True` if result is available. The third way to achieve the same is to call `TryValue` periodically. `TryValue` also waits some specified number of milliseconds and returns `True` if result is available but in addition it will also return the result in the value parameter.

The fourth and completely different way is to specify a termination handler which will notify you when the background calculation is completed. The following example sets the termination handler to get the value of the background calculation into the memo field and then destroy the Future interface.

```
procedure StartHttpGet(const url: string);
begin
  FGetFuture := Parallel.Future<string>(
    function: string
    begin
      Result := HttpGet(url);
    end,
    Parallel.TaskConfig.OnTerminated(
      procedure
      begin
        Memo1.Text := FGetFuture.Value;
        FGetFuture := nil;
      end
    )
  );
end;
```

3.6.3 Cancellation

It is possible to cancel the background execution of the Future before it is completed. The Future uses Cancellation token mechanism to achieve this. Cancellation is cooperative – if the background task does not willingly cancel itself, cancellation will fail.

To cancel a background task, the Future owner (the code that called `Parallel.Future`) has to call the `Cancel` method on the `IOmniFuture<T>` interface. This will signal the cancellation token which the background task must check periodically. To get access to the cancellation token, background code must be declared as a function accepting an `IOmniTask` parameter.

The following (pretty much pointless) program illustrates this concept.

```
var
  FCountFuture: IOmniFuture<integer>;

function CountTo100(const task: IOmniTask): integer;
var
  i: integer;
begin
  for i := 1 to 100 do begin
    Sleep(100);
    Result := i;
    if task.CancellationToken.IsSignalled then
      break; //for
  end;
end;

procedure StartCounting;
begin
  FCountFuture := Parallel.Future<integer>(CountTo100);
  Sleep(100);
  FCountFuture.Cancel;
  FCountFuture.WaitFor(INFINITE);
  FCountFuture := nil;
end;
```

`StartCounting` creates a Future which executes `CountTo100` function in the background. It then sleeps 100 milliseconds, calls the `Cancel` function, waits for the Future to terminate and clears the Future interface.

`CountTo100` function counts from 1 to 100. It sleeps for 100 milliseconds after each number, stores the current counter in the function result and then checks the cancellation token. If it is signaled (meaning that the owner called the `Cancel` function), it will break out of the for loop.

If you put a breakpoint on the last line of the `StartCounting` function and run the program,

you'll see that it will be reached almost immediately, proving that the `CountTo100` did not take 10 seconds to return a result (100 repeats * 100 milliseconds = 10 seconds).

You cannot call the `Value` function if the calculation was cancelled as it would raise an `EFutureCancelled` exception. If you don't know whether the `Cancel` was called or not, you can call the `IOmniFuture<T>.IsCancelled` and check the result (`True` = calculation was cancelled).

3.6.4 Handling exceptions

If the background code raises unhandled exception (i.e. the exception was not captured in a `try..except` block), OmniThreadLibrary catches this exception and gracefully completes the background task. When you call the `Value` function, this exception is re-raised.

This immensely helps with debugging as the background exceptions (exceptions in background threads) are ignored when you run the program without a debugger. By default Delphi does nothing with them – it behaves as if nothing is wrong and that can be quite dangerous. As the Future exceptions are re-raised in the main thread when the `Value` is called this makes them equivalent to other exceptions in the main thread.

There are few different ways to handle exceptions in Future and they are most simply explained through the code. First example catches the exception by wrapping the `Value` call in `try..except`.

```
1  procedure FutureException1;
2  var
3    future: IOmniFuture<integer>;
4  begin
5    future := Parallel.Future<integer>(
6      function: integer
7      begin
8        raise ETestException.Create('Exception in Parallel.Future');
9      end
10   );
11   Log('Future is executing ...');
12   Sleep(1000);
13   try
14     Log('Future retured: %d', [future.Value]);
15   except
16     on E: Exception do
17       Log('Future raised exception %s:%s', [E.ClassName, E.Message]);
18   end;
19 end;
```

Second example uses `WaitFor` to wait on task completion and then checks the result of the `FatalException` function. It will return `Nil` if there was no exception or the exception object if there was an exception. Exception object itself will still be owned by the Future task and will be destroyed when the Future is destroyed.

```
1  procedure FutureException2;
2  var
3    future: IOmniFuture<integer>;
4  begin
5    future := Parallel.Future<integer>(
6      function: integer
7      begin
8        raise ETestException.Create('Exception in Parallel.Future');
9      end
10   );
11   Log('Future is executing ...');
12   future.WaitFor(INFINITE);
13   if assigned(future.FatalException) then
14     Log('Future raised exception %s:%s',
15       [future.FatalException.ClassName, future.FatalException.Message])
```

```
  else
    Log('Future retured: %d', [future.Value]);
end;
```

Third example shows how you can detach exception from the future. By calling `DetachException` you will get the ownership of the exception object and you should destroy it at some appropriate point in time.

```
procedure FutureException3;
var
  excFuture: Exception;
  future   : IOmniFuture<integer>;
begin
  future := Parallel.Future<integer>(
    function: integer
    begin
      raise ETestException.Create('Exception in Parallel.Future');
    end
  );
  Log('Future is executing ...');
  future.WaitFor(INFINITE);
  excFuture := future.DetachException;
  try
    if assigned(excFuture) then
      Log('Future raised exception %s:%s',
        [excFuture.ClassName, excFuture.Message])
    else
      Log('Future retured: %d', [future.Value]);
  finally FreeAndNil(excFuture); end;
end;
```

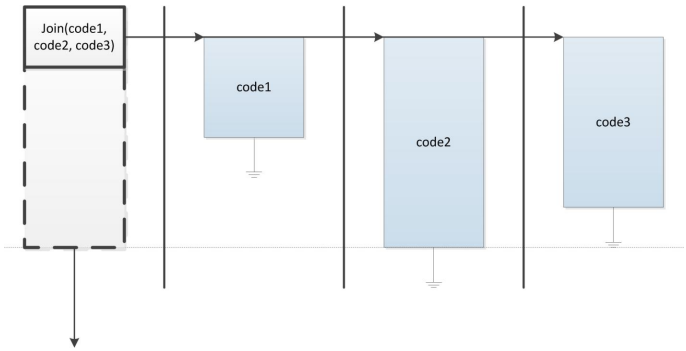
See also demo `48_OtlParallelExceptions`.

3.6.5 Examples

Practical examples of Future usage can be found in chapter OmniThreadLibrary and COM/OLE.

3.7 Join

Join abstraction enables you to start multiple background tasks in multiple threads. To create the task, call `Parallel.Join`.



When you use Parallel.Join, background tasks are started in background threads. By default, Join waits for all background threads to complete before the control is returned to the caller.

See also demo `37_ParallelJoin`.

Example:

```
Parallel.Join(
  procedure
  var
    i: integer;
  begin
    for i := 1 to 8 do begin
      Sleep(200);
      MessageBeep($FFFFFFFF);
    end,
  procedure
  var
    i: integer;
  begin
    for i := 1 to 10 do begin
      Sleep(160);
      MessageBeep($FFFFFFFF);
    end;
  end
).Execute;
```

This simple program executes two background tasks, each beeping at different frequency. Each task is coded as an anonymous method but you can also use a normal method or a normal procedure for the task code.

The `Parallel` class defines five `Join` overloads. The first creates empty `IOmniParallelJoin` interface. Next two create the same interface but configured with two tasks and the last two create this interface configured for any number of tasks. Tasks can be of two different types – parameterless methods and methods containing one parameter of the `IOmniJoinState` type.

```
type
  TOmniJoinDelegate = reference to procedure (const joinState:
                        IOmniJoinState);

  Parallel = class
    class function Join: IOmniParallelJoin; overload;
    class function Join(const task1, task2: TProc): IOmniParallelJoin;
                     overload;
    class function Join(const task1, task2: TOmniJoinDelegate):
                     IOmniParallelJoin; overload;
    class function Join(const tasks: array of TProc): IOmniParallelJoin;
                     overload;
    class function Join(const tasks: array of TOmniJoinDelegate):
                     IOmniParallelJoin;overload;
    ...
  end;
```

3.7.1 IOmniParallelJoin interface

`Parallel.Join` returns an `IOmniParallelJoin` interface which you can use to specify tasks, start and control execution and handle exceptions.

```
type
  IOmniParallelJoin = interface
    function Cancel: IOmniParallelJoin;
    function DetachException: Exception;
    function Execute: IOmniParallelJoin;
    function FatalException: Exception;
    function IsCancelled: boolean;
    function IsExceptional: boolean;
    function NumTasks(numTasks: integer): IOmniParallelJoin;
    function OnStop(const stopCode: TProc): IOmniParallelJoin; overload;
```

```
    function OnStop(const stopCode: TOmniTaskStopDelegate): IOmniParallelJoin; overload;
    function OnStopInvoke(const stopCode: TProc): IOmniParallelJoin;
    function Task(const task: TProc): IOmniParallelJoin; overload;
    function Task(const task: TOmniJoinDelegate): IOmniParallelJoin;              overload;
    function TaskConfig(const config: IOmniTaskConfig): IOmniParallelJoin;
    function NoWait: IOmniParallelJoin;
    function WaitFor(timeout_ms: cardinal): boolean;
  end;
```

The most important of these functions is `Execute`. It will start appropriate number of background threads and start executing tasks in those threads. By default, Join uses as many threads as there are tasks but you can override this behaviour by calling the `NumTasks` function.

If `NumTasks` receives a positive parameter (`> 0`), the number of worker tasks is set to that number. For example, `NumTasks(16)` starts 16 worker tasks, even if that is more than the number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies number of cores that should be reserved for other use. The number of worker tasks is then set to `<number of available cores> - <number of reserved cores>`. If, for example, current process can use 16 cores and `NumTasks(-4)` is used, only 12 (916-4*) worker tasks will be started.

Value 0 is not allowed and results in an exception.

You can add tasks to a Join abstractions by calling the `Task` function. In fact, that's just how the `Parallel.Join` overloads are implemented.

```
class function Parallel.Join(const task1, task2: TProc): IOmniParallelJoin;
begin
  Result := TOmniParallelJoin.Create.Task(task1).Task(task2);
end;

class function Parallel.Join(const tasks: array of TProc): IOmniParallelJoin;
var
  aTask: TProc;
begin
  Result := TOmniParallelJoin.Create;
  for aTask in tasks do
    Result.Task(aTask);
end;
```

To set up task configuration block, call the `TaskConfig` function. Following example uses `TaskConfig` to set up critical section which is then used in two parallel tasks to protect the shared resource. Workers use the `IOmniJoinState` instance to access the `IOmniTask` interface and through it the `Lock` property.

```
FSharedValue := 42;
Parallel.Join(
  procedure (const joinState: IOmniJoinState)
  var
    i: integer;
  begin
    for i := 1 to 1000000 do begin
      joinState.Task.Lock.Acquire;
      FSharedValue := FSharedValue + 17;
      joinState.Task.Lock.Release;
    end;
  end,
  procedure (const joinState: IOmniJoinState)
  var
    i: integer;
```

```
  begin
    for i := 1 to 1000000 do begin
      joinState.Task.Lock.Acquire;
      FSharedValue := FSharedValue - 17;
      joinState.Task.Lock.Release;
    end;
  end
).TaskConfig(Parallel.TaskConfig.WithLock(CreateOmniCriticalSection))
 .Execute;
```

By default, `Join` will wait for all background tasks to complete execution. Alternatively, you can call the `NoWait` function, after which `Join` will just start the tasks and return immediately. If you want to be notified when all tasks are finished, you can assign the termination handler by calling the `OnStop` function. This termination handler is called from one of the worker threads, not from the main thread! If you need to run a code in the main thread, use the task configuration block.

Release [3.07.2] introduced method `OnStopInvoke` which works like `OnStop` except that the termination handler is automatically executed in the context of the owner thread via implicit `Invoke`. For example, see `Parallel.ForEach.OnStopInvoke`.

You can also call `WaitFor` to wait for the `Join` to finish. `WaitFor` accepts an optional timeout parameter; by default it will wait as long as needed.

3.7.2 IOmniJoinState interface

Task method can accept a parameter of type `IOmniJoinState`. This allows it to access the `IOmniTask` interface, participate in the cooperative cancellation and check for exceptions.

```
type
  IOmniJoinState = interface
    function  GetTask: IOmniTask;
  //
    procedure Cancel;
    function  IsCancelled: boolean;
    function  IsExceptional: boolean;
    property Task: IOmniTask read GetTask;
  end;
```

3.7.3 Cancellation

Join background tasks support cooperative cancellation. If you are using `TOmniJoinDelegate` tasks (that is, tasks accepting the IOmniJoinState parameter), any task can call the `Cancel` method of this interface. This, in turn, sets internal cancellation flag which may be queried by calling the `IsCancelled` method. That way, one task can interrupt other tasks provided that they are testing `IsCancelled` repeatedly.

Main thread can also cancel its subtasks (when using `NoWait`) by calling `IOmniParallelJoin.Cancel` and can test the cancellation flag by calling `IsCancelled`.

The following demo code demonstrates most of concepts mentioned above.

```
var
  join: IOmniParallelJoin;
  time: int64;
begin
  FJoinCount.Value := 0;
  FJoinCount2.Value := 0;
  join := Parallel.Join(
    procedure (const joinState: IOmniJoinState)
```

```
    var
      i: integer;
    begin
      for i := 1 to 10 do begin
        Sleep(100);
        FJoinCount.Increment;
        if joinState.IsCancelled then
          break; //for
      end;
    end,
    procedure (const joinState: IOmniJoinState)
    var
      i: integer;
    begin
      for i := 1 to 10 do begin
        Sleep(200);
        FJoinCount2.Increment;
        if joinState.IsCancelled then
          break; //for
      end;
    end
  ).NoWait.Execute;
  Sleep(500);
  time := DSiTimeGetTime64;
  join.Cancel.WaitFor(INFINITE);
  Log(Format('Waited %d ms for joins to terminate',
    [DSiElapsedTime64(time)]));
  Log(Format('Tasks counted up to %d and %d',
    [FJoinCount.Value, FJoinCount2.Value]));
end;
```

The call to `Parallel.Join` starts two tasks. Because the `NoWait` is used, the call returns immediately and stores resulting `IOmniParallelJoin` interface in the local variable `join`. Main code then sleeps for half a second, cancels the execution and waits for background tasks to terminate.

Both tasks execute a simple loop which waits a little, increments a counter and checks the cancellation flag. Because the cancellation flag is set after 500 ms, we would expect five or six repetitions of the first loop (five repetitions take exactly 500 ms and we can't tell exactly what will execute first – `Cancel` or fifth `IsCancelled`) and three repetitions of the second loop. That is exactly what the program prints out.

3.7.4 Handling exceptions

Exceptions in background tasks are caught and re-raised in the `WaitFor` method. If you are using synchronous version of Join (without the `NoWait` modifier), then `WaitFor` is called at the end of the `Execute` method (in other words, `Parallel.Join(…).Execute` will re-raise task exceptions). If, however, you are using the asynchronous version (by calling `Parallel.Join(…).NoWait.Execute`), exception will only be raised when you wait for the background tasks to complete by calling `WaitFor`.

You can test for the exception by calling the `FatalException` function. It will first wait for all background tasks to complete (without raising the exception) and then return the exception object. You can also detach the exception object from the Join and handle it yourself by using the `DetachException` function.

There's also an `IsExceptional` function (available in `IOmniParallelJoin` and `IOmniJoinState` interfaces) which tells you whether any background task has thrown an exception.

As Join executes multiple tasks, there can be multiple background exceptions. To get you full access to those exceptions, Join wraps them into `EJoinException` object.

```
type
  EJoinException = class(Exception)
    constructor Create; reintroduce;
    destructor  Destroy; override;
    procedure Add(iTask: integer; taskException: Exception);
    function  Count: integer;
    property Inner[idxException: integer]: TJoinInnerException
      read GetInner; default;
  end;
```

This exception class contains combined error messages from all background tasks in its `Message` property and allows you to access exception information for all caught exceptions directly with the `Inner[]` property. The following code demonstrates this.

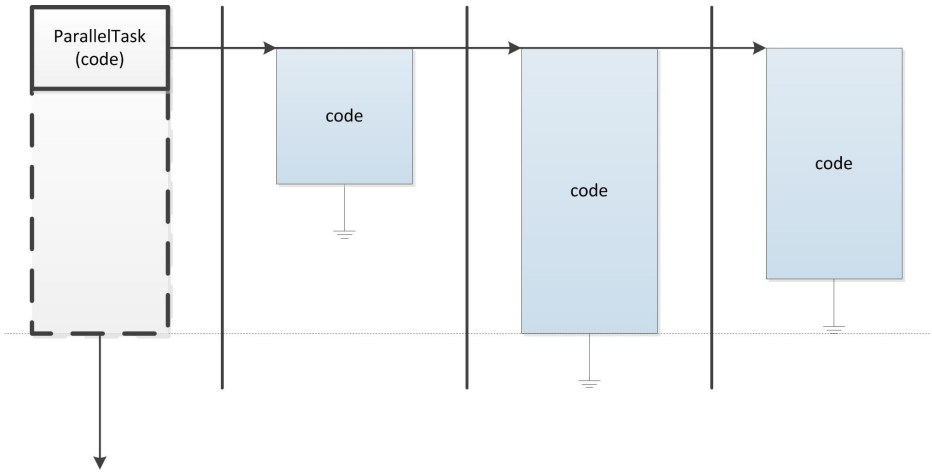
```
var
  iInnerExc: integer;
begin
  try
    Parallel.Join([
      procedure begin
        raise ETestException.Create('Exception 1 in Parallel.Join');
      end,
      procedure begin
      end,
      procedure begin
        raise ETestException.Create('Exception 2 in Parallel.Join');
      end]).Execute;
  except
    on E: EJoinException do begin
      Log('Join raised exception %s:%s', [E.ClassName, E.Message]);
      for iInnerExc := 0 to E.Count - 1 do
        Log(' Task #%d raised exception: %s:%s', [E[iInnerExc].TaskNumber,
          E[iInnerExc].FatalException.ClassName,
          E[iInnerExc].FatalException.Message]);
    end;
  end;
end;
```

The `iInnerExc` variable loops over all caught exceptions and for each such exception displays the task number (starting with 0), exception class and exception message. This approach allows you to log the exception or, if you are interested in details, examine specific inner exceptions and handle them appropriately.

See also demo `48_OtlParallelExceptions`.

3.8 Parallel task

Parallel task abstraction enables you to start the same method in multiple threads. To create the task, call `Parallel.ParallelTask`.



When you use Parallel.ParallelTask, same background task is started in multiple background threads. By default, ParallelTask waits for all background threads to complete before the control is returned to the caller.

Example:

```
Parallel.ParallelTask.NumTasks(3).Execute(
  procedure
  begin
    while true do
      ;
  end
);
```

This simple code fragment starts infinite loop in three threads. The task is coded as an anonymous method but you can also use a normal method or a normal procedure for the task code.

The `Parallel` class implements function `ParallelTask` which returns an `IOmniParallelTask` interface. All configuration of the parallel task is done via this interface.

```
type
  Parallel = class
    class function ParallelTask: IOmniParallelTask;
    ...
  end;
```

3.8.1 IOmniParallelTask interface

```
type
  IOmniParallelTask = interface
    function  Execute(const aTask: TProc): IOmniParallelTask; overload;
    function  Execute(
      const aTask: TOmniParallelTaskDelegate): IOmniParallelTask; overload;
    function  NoWait: IOmniParallelTask;
    function  NumTasks(numTasks: integer): IOmniParallelTask;
    function  OnStop(const stopCode: TProc): IOmniParallelTask; overload;
    function  OnStop(const stopCode: TOmniTaskStopDelegate): IOmniParallelTask; overload;
    function  OnStopInvoke(const stopCode: TProc): IOmniParallelTask;
    function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelTask;
    function  WaitFor(timeout_ms: cardinal): boolean;
  end;
```

The most important of these functions is `Execute`. It will start appropriate number of background threads and start executing task in those threads.

There are two overloaded versions of `Execute`. The first accepts a parameter-less background

task and the second accepts a background task with an IOmniTask parameter.

By default, Parallel task uses as many threads as there are tasks but you can override this behaviour by calling the NumTasks function.

If NumTasks receives a positive parameter (> 0), the number of worker tasks is set to that number. For example, NumTasks(16) starts 16 worker tasks, even if that is more than the number of available cores.

If NumTasks receives a negative parameter (< 0), it specifies number of cores that should be reserved for other use. The number of worker tasks is then set to <number of available cores> - <number of reserved cores>. If, for example, current process can use 16 cores and NumTasks(-4) is used, only 12 (16-4) worker tasks will be started.

Parameter 0 is not allowed and results in an exception.

The OnStop function can be used to set up a termination handler - a code that will get executed when all background tasks will have completed execution. This termination handler is called from one of the worker threads, not from the main thread! If you need to run a code in the main thread, use the task configuration block. To set up task configuration block, call the TaskConfig function.

Release [3.07.2] introduced method OnStopInvoke which works like OnStop except that the termination handler is automatically executed in the context of the owner thread via implicit Invoke. For example, see Parallel.ForEach.OnStopInvoke.

A call to the WaitFor function will wait for up to timeout_ms milliseconds (this value can be set to INFINITE) for all background tasks to terminate. If the tasks terminate in the specified time, WaitFor will return True. Otherwise, it will return False.

3.8.2 Example

The following code uses ParallelTask to generate large quantities of pseudorandom data. The data is written to an output stream.

```
1  procedure CreateRandomFile(fileSize: integer; output: TStream);
2  const
3    CBlockSize = 1 * 1024 * 1024 {1 MB};
4  var
5    buffer   : TOmniValue;
6    memStr   : TMemoryStream;
7    outQueue : IOmniBlockingCollection;
8    unwritten: IOmniCounter;
9  begin
10   outQueue := TOmniBlockingCollection.Create;
11   unwritten := CreateCounter(fileSize);
12   Parallel.ParallelTask.NoWait
13     .NumTasks(Environment.Process.Affinity.Count)
14     .OnStop(Parallel.CompleteQueue(outQueue))
15     .Execute(
16       procedure
17       var
18         buffer      : TMemoryStream;
19         bytesToWrite: integer;
20         randomGen   : TGpRandom;
21       begin
22         randomGen := TGpRandom.Create;
23         try
24           while unwritten.Take(CBlockSize, bytesToWrite) do begin
25             buffer := TMemoryStream.Create;
26             buffer.Size := bytesToWrite;
```

```
27            FillBuffer(buffer.Memory, bytesToWrite, randomGen);
28            outQueue.Add(buffer);
29          end;
30        finally FreeAndNil(randomGen); end;
31      end
32    );
33  for buffer in outQueue do begin
34    memStr := buffer.AsObject as TMemoryStream;
35    output.CopyFrom(memStr, 0);
36    FreeAndNil(memStr);
37  end;
38 end;
```

The code creates a blocking collection to hold buffers with pseudorandom data. Then it creates a counter which will hold the count of bytes that have yet to be written.

`ParallelTask` is used to start parallel workers. Each worker initializes its own pseudorandom data generator and then keeps generating buffers with pseudorandom data until the counter drops to zero. Each buffer is written to the blocking collection.

Because of the `NoWait` modifier, main thread continues with the execution immediately after all threads have been scheduled. Main thread keeps reading buffers from the blocking collection and writes the content of those buffers into the output stream. (As the `TStream` is not thread-safe, we cannot write to the output stream directly from multiple background threads.)

`For..in` loop will block if there is no data in the blocking collection, but it will only stop looping after the blocking collection's `CompleteAdding` method is called. This is done with the help of the `Parallel.CompleteQueue` helper which is called from the termination handler (`OnStop`).

```
1 class function Parallel.CompleteQueue(
2   const queue: IOmniBlockingCollection): TProc;
3 begin
4   Result :=
5     procedure
6     begin
7       queue.CompleteAdding;
8     end;
9 end;
```

3.8.3 Handling exceptions

Exceptions in background tasks are caught and re-raised in the `WaitFor` method. If you are using synchronous version of Parallel task (without the `NoWait` modifier), then `WaitFor` is called at the end of the `Execute` method (in other words, `Parallel.ParallelTask.Execute(...)` will re-raise task exceptions). If, however, you are using the asynchronous version (by calling `Parallel.Parallelltask.NoWait.Execute(...)`), exception will only be raised when you wait for the background tasks to complete by calling `WaitFor`.

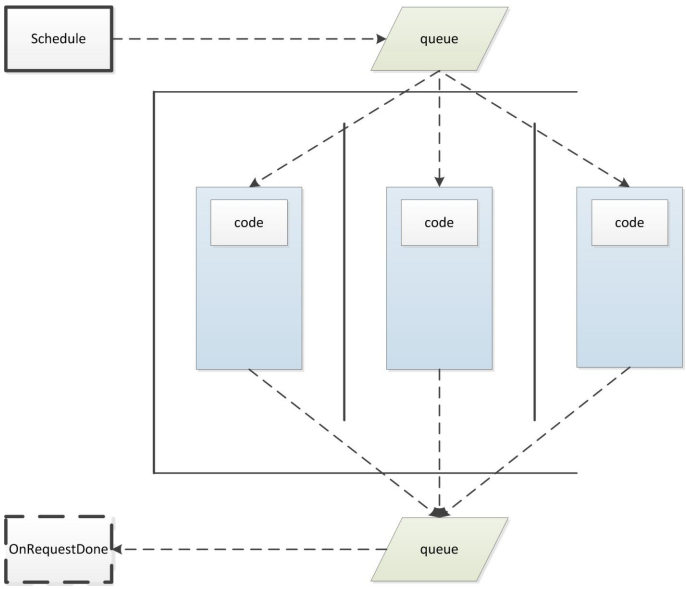
For more details on handling ParallelTask exceptions, see the Handling exceptions section in the Join chapter.

3.8.4 Examples

Practical examples of Parallel Task usage can be found in chapters Background worker and list partitioning and Parallel data production.

3.9 Background worker

Background worker abstraction implements a client/server relationship. To create a Background worker abstraction, call `Parallel.BackgroundWorker`.



When you call `Parallel.BackgroundWorker.Execute`, OmniThreadLibrary starts your task in one or more threads. Task code is wrapped in a wrapper which receives requests from the owner, calls the task to process request and returns result to the owner.

See also demo `52_BackgroundWorker`.

Example:

Start the worker

```
FBackgroundWorker := Parallel.BackgroundWorker.NumTasks(2)
  .Execute(
    procedure (const workItem: IOmniWorkItem)
    begin
      workItem.Result := workItem.Data.AsInteger * 3;
    end
  )
  .OnRequestDone(
    procedure (const Sender: IOmniBackgroundWorker;
      const workItem: IOmniWorkItem)
    begin
      lbLogBW.ItemIndex := lbLogBW.Items.Add(Format('%d * 3 = %d',
        [workItem.Data.AsInteger, workItem.Result.AsInteger]));
    end
  );
```

Schedule a work item

```
FBackgroundWorker.Schedule(
  FBackgroundWorker.CreateWorkItem(Random(100)));
```

Stop the worker

```
FBackgroundWorker.Terminate(INFINITE);
FBackgroundWorker := nil;
```

3.9.1 Basics

Background worker is designed around the concept of a work item. You create a worker, which spawns one or more background threads, and then schedule work items to it. When they are processed, the abstraction notifies you so you can process the result. Work items are queued so you can schedule many work items at once and background threads will then process them one by one.

Background worker is created by calling `Parallel.BackgroundWorker` factory. Usually you'll also set the main work item processing method and completion method by calling `Execute` and `OnRequestDone`, respectively. As usual, you can provide OTL with a method, a procedure, or an anonymous method.

```
1 FWorker := Parallel.BackgroundWorker
2   .OnRequestDone(StringRequestDone)
3   .Execute(StringProcessorHL);
```

To close the background worker, call the `Terminate` method and set the reference (`FWorker`) to `nil`.

To create a work item, call the `CreateWorkItem` factory and pass the result to the `Schedule` method. You can pass any data to the work item by passing a parameter to the `CreateWorkItem` method. If you have to pass multiple parameters, you can collect them in a record and wrap it with a `TOmniValue.FromRecord<T>`, pass them as an array of `TOmniValue`s or pass them as an object or an interface.

3.9.2 IOmniBackgroundWorker interface

```
1  type
2    TOmniTaskInitializerDelegate =
3      reference to procedure(var taskState: TOmniValue);
4    TOmniTaskFinalizerDelegate =
5      reference to procedure(const taskState: TOmniValue);
6
7    IOmniBackgroundWorker = interface
8      function  CreateWorkItem(const data: TOmniValue): IOmniWorkItem;
9      procedure CancelAll; overload;
10     procedure CancelAll(upToUniqueID: int64); overload;
11     function  Config: IOmniWorkItemConfig;
12     function  Execute(const aTask: TOmniBackgroundWorkerDelegate = nil):
13       IOmniBackgroundWorker;
14     function  Finalize(taskFinalizer:
15       TOmniTaskFinalizerDelegate): IOmniBackgroundWorker;
16     function  Initialize(taskInitializer:
17       TOmniTaskInitializerDelegate): IOmniBackgroundWorker;
18     function  NumTasks(numTasks: integer): IOmniBackgroundWorker;
19     function  OnRequestDone(const aTask: TOmniWorkItemDoneDelegate):
20       IOmniBackgroundWorker;
21     function  OnRequestDone_Asy(const aTask: TOmniWorkItemDoneDelegate):
22       IOmniBackgroundWorker;
23     function  OnStop(stopCode: TProc): IOmniBackgroundWorker; overload;
24     function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniBackgroundWorker; overload;
25     function  OnStopInvoke(stopCode: TProc): IOmniBackgroundWorker;
26     procedure Schedule(const workItem: IOmniWorkItem;
27       const workItemConfig: IOmniWorkItemConfig = nil);
28     function  TaskConfig(const config: IOmniTaskConfig):
29       IOmniBackgroundWorker;
30     function  Terminate(maxWait_ms: cardinal): boolean;
31     function  WaitFor(maxWait_ms: cardinal): boolean;
32   end;
```

Background worker supports two notification mechanisms. By calling `OnRequestDone`, you are setting a synchronous handler, which will be executed in the context of the thread that created the background worker (usually a main thread). In other words – if you call

OnRequestDone, you don't have to worry about thread synchronisation issues. On the other hand, OnRequestDone_Asy handler is executed asynchronously, in the context of the thread that processed the work item.

For performance reasons (for example when terminating the application), the code can prevent execution of event handlers for a work item by setting IOmniWorkItem.SkipCompletionHandler to True.

By calling NumTasks, you can set the degree of parallelism. By default, background worker uses only one background task but you can override this behaviour.

If NumTasks receives a positive parameter (> 0), the number of worker tasks is set to that number. For example, NumTasks(16) starts 16 worker tasks, even if that is more then the number of available cores.

If NumTasks receives a negative parameter (< 0), it specifies number of cores that should be reserved for other use. The number of worker task is then set to <number of available cores> - <number of reserved cores>. If, for example, current process can use 16 cores and NumTasks(-4) is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

The OnStop function can be used to set up a termination handler - a code that will get executed when all background tasks will have completed execution. This termination

handler is called from one of the worker threads, not from the main thread! If you need to run a code in the main thread, use the task configuration block. To set up task configuration block, call the TaskConfig function.

Release [3.07.2] introduced method OnStopInvoke which works like OnStop except that the termination handler is automatically executed in the context of the owner thread via implicit Invoke. For example, see Parallel.ForEach.OnStopInvoke.

Calling Terminate will stop background workers. If they stop in maxWait_ms, True will be returned, False otherwise. WaitFor waits for workers to stop (without commanding them to stop beforehand so you would have to call Terminate before WaitFor) and returns True/False just as Terminate does.

3.9.3 Task initialization

Background worker implements mechanism that can be used by worker tasks to intialize and destroy task-specific structures.

By calling Initialize you can provide a task initializer which is executed once for each worker task before it begins processing work items. Similarly, by calling Finalize you provide the background worker with a task finalizer which is called just before the background task is destroyed.

Both initializer and finalizer will receive a taskState variable where you can store any task-specific data (for example, a class containing multiple task-specific structures). This task state is also available in the executor method throught the property IOmniWorkItem.TaskState.

3.9.4 Work item configuration

You can pass additional configuration parameters to the Schedule method by providing a configuration block, which can be created by calling the Config method. By using this

approach, you can set a custom executor method or a custom completion method for each separate work item.

```
type
  IOmniWorkItemConfig = interface
    function  OnExecute(const aTask: TOmniBackgroundWorkerDelegate):
      IOmniWorkItemConfig;
    function  OnRequestDone(const aTask: TOmniWorkItemDoneDelegate):
      IOmniWorkItemConfig;
    function  OnRequestDone_Asy(const aTask: TOmniWorkItemDoneDelegate):
      IOmniWorkItemConfig;
  end;
```

3.9.5 Work item interface

`CreateWorkItem` method returns an `IOmniWorkItem` interface.

```
type
  IOmniWorkItem = interface
    function  DetachException: Exception;
    function  FatalException: Exception;
    function  IsExceptional: boolean;
    property CancellationToken: IOmniCancellationToken
      read GetCancellationToken;
    property Data: TOmniValue read GetData;
    property Result: TOmniValue read GetResult write SetResult;
    property SkipCompletionHandler: boolean read GetSkipCompletionHandler write
      SetSkipCompletionHandler;
    property Task: IOmniTask read GetTask;
    property TaskState: TOmniValue read GetTaskState;
    property UniqueID: int64 read GetUniqueID;
  end;
```

It contains input data (`Data` property), result (`Result` property) and a unique ID, which is assigned in the `CreateWorkItem` call. First work item gets ID 1, second ID 2 and so on. This allows for some flexibility when you want to cancel work items. You can cancel one specific item by calling `workItem.CancellationToken.Signal` or multiple items by calling `backgroundWorker.CancelAll(highestIDToBeCancelled)` or all items by calling `backgroundWorker.CancelAll`.

Cancellation is partly automatic and partly cooperative. If the work item that is to be cancelled has not yet reached the execution, the system will prevent it from ever being executed. If, however, work item is already being processed, your code must occasionally check `workItem.CancellationToken.IsSignalled` and exit if that happens (provided that you want to support cancellation at all). Regardless of how the work item was cancelled, completion handler will still be called and it can check `workItem.CancellationToken.IsSignalled` to check whether the work item was cancelled prematurely or not.

Any uncaught exception will be stored in the `FatalException` property. You can detach (and take ownership of) the exception by calling the `DetachException` and you can test if there was an exception by calling `IsExceptional`. If `IsExceptional` returns `True`, any access to the `Result` property will raise exception stored in the `FatalException` property. [In other words – if an unhandled exception occurs in the executor code (in the background thread), it will propagate to the place where you access `workItem.Result`.]

Property `Task` provides access to the task executing the work item. Property `TaskState` returns the value initialized in the task initializer.

If `SkipCompletionHandler` is set to `True` when work item is created or during its execution, request handlers for that work item won't be called.

If `SkipCompletionHandler` is set to `True` in the `OnRequestDone_Asy` handler, then `OnRequestDone` handler

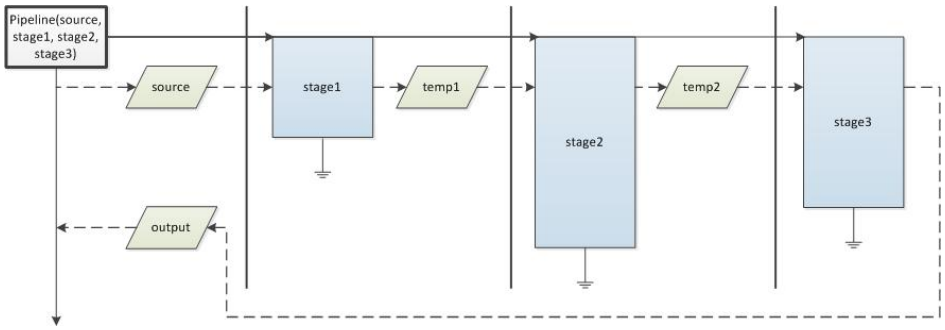
won't be called.

3.9.6 Examples

Practical examples of Background Worker usage can be found in chapters Background worker and list partitioning and OmniThreadLibrary and databases.

3.10 Pipeline

Pipeline abstraction encapsulates multistage processes in which the data processing can be split into multiple independent stages connected with data queues.



When you call `Parallel.Pipeline`, background tasks for all stages are started each in its own thread. The system also creates input queue, output queue and interconnecting queues.

See also demo `41_Pipeline`.

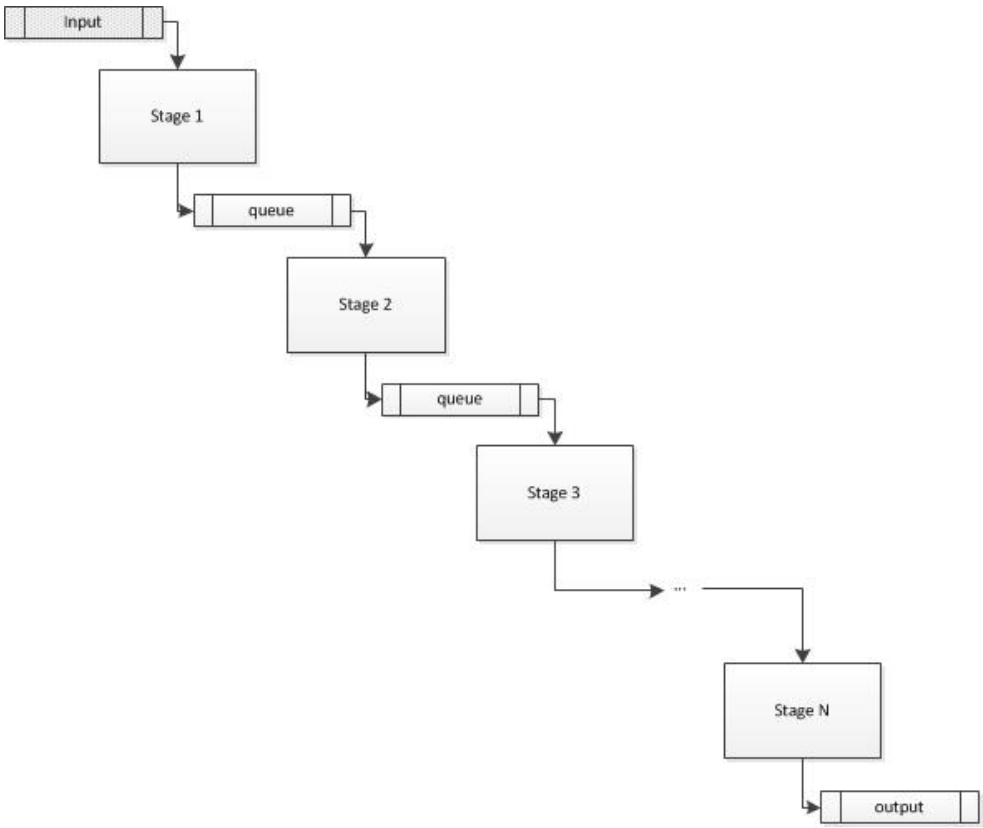
Example:

```
1  uses
2    OtlCommon,
3    OtlCollections,
4    OtlParallel;
5
6  var
7    sum: integer;
8
9  sum := Parallel.Pipeline
10   .Stage(
11     procedure (const input, output: IOmniBlockingCollection)
12     var
13       i: integer;
14     begin
15       for i := 1 to 1000000 do
16         output.Add(i);
17     end)
18   .Stage(
19     procedure (const input: TOmniValue; var output: TOmniValue)
20     begin
21       output := input.AsInteger * 3;
22     end)
23   .Stage(
24     procedure (const input, output: IOmniBlockingCollection)
25     var
26       sum: integer;
27       value: TOmniValue;
28     begin
29       sum := 0;
30       for value in input do
31         Inc(sum, value);
32       output.Add(sum);
33     end)
34   .Run.Output.Next;
```

This example creates a three-stage pipeline. First stage generates numbers from 1 to 1.000,000. Second stage multiplies each number by 3. Third stage adds all numbers together and writes result to the output queue. This result is then stored in the variable `sum`.

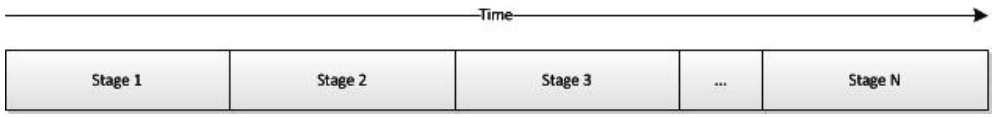
3.10.1 Background

The Pipeline abstraction is appropriate for parallelizing processes that can be split into stages (subprocesses), connected with data queues. Data flows from the input queue into the first stage, where it is partially processed and then emitted into an intermediary queue. The first stage then continues execution, processes more input data and outputs more output data. This continues until complete input is processed. The intermediary queue leads into the next stage which does the processing in a similar manner and so on and on. At the end, the data is output into a queue which can be then read and processed by the program that created this multistage process. As a whole, a multistage process acts as a pipeline – data comes in, data comes out (and a miracle occurs in-between ;)).

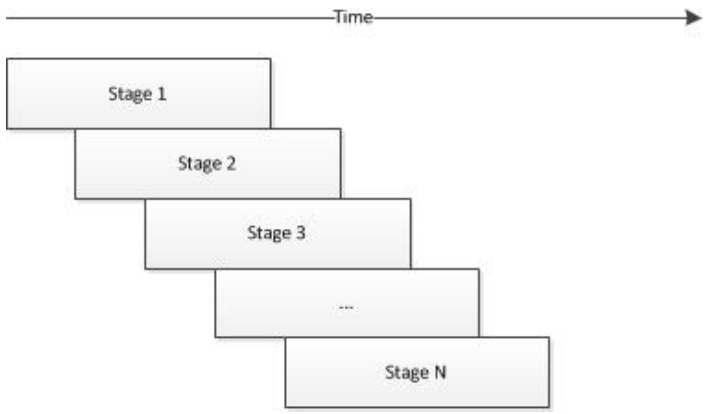


What is important here is that no stage shares state with any other stage. The only interaction between stages is done with the data passed through the intermediary queues. The quantity of data, however, doesn’ t have to be constant. It is entirely possible for a stage to generate more or less data than it received on the input queue.

In a classical single-threaded program the execution plan for a multistage process is very simple.



In a multithreaded environment, however, we can do better than that. Because the stages are largely independent, they can be executed in parallel.



3.10.2 Basics

A Pipeline is created by calling `Parallel.Pipeline` function which returns an `IOmniPipeline` interface. There are three overloaded versions of this function. The first creates an unconfigured pipeline. The second prepares one or more stages and optionally sets the input queue. The third prepares one or more stages with a different method signature.

```
1 TPipelineStageDelegate = reference to procedure (const input, output:
2   IOmniBlockingCollection);
3 TPipelineStageDelegateEx = reference to procedure (const input, output:
4   IOmniBlockingCollection; const task: IOmniTask);
5
6 class function Pipeline: IOmniPipeline; overload;
7 class function Pipeline(const stages: array of TPipelineStageDelegate;
8   const input: IOmniBlockingCollection = nil): IOmniPipeline; overload;
9 class function Pipeline(const stages: array of TPipelineStageDelegateEx;
10  const input: IOmniBlockingCollection = nil): IOmniPipeline; overload;
```

Stages are implemented as anonymous procedures, procedures or methods taking two queue parameters – one for input and one for output. Both queues are automatically created by the Pipeline implementation and passed to the stage delegate.

Pipeline also supports concept of simple stages where stage method accepts a `TOmniValue` input and provides a `TOmniValue` output. In this case, OmniThreadLibrary provides the loop which reads data from the input queue, calls your stage code and writes data to the output queue.

```
1 TPipelineSimpleStageDelegate = reference to procedure(
2   const input: TOmniValue; var output: TOmniValue);
```

Simple stage can produce zero or one data element for each input. Only if the code assigns a value to the `output` parameter, the value will be written to the output queue.

3.10.3 IOmniPipeline interface

All `Parallel.Pipeline` overloads return the `IOmniPipeline` interface.

```
1 IOmniPipeline = interface
2   procedure Cancel;
3   function  From(const queue: IOmniBlockingCollection): IOmniPipeline;
4   function  HandleExceptions: IOmniPipeline;
5   function  NoThrottling: IOmniPipeline;
6   function  NumTasks(numTasks: integer): IOmniPipeline;
7   function  OnStop(stopCode: TProc): IOmniPipeline; overload;
8   function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniPipeline; overload;
9   function  OnStopInvoke(stopCode: TProc): IOmniPipeline;
10  function  Run: IOmniPipeline;
11  function  Stage(
```

```
    pipelineStage: TPipelineSimpleStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stage(
    pipelineStage: TPipelineStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stage(
    pipelineStage: TPipelineStageDelegateEx;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineSimpleStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineStageDelegate;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Stages(
    const pipelineStages: array of TPipelineStageDelegateEx;
    taskConfig: IOmniTaskConfig = nil): IOmniPipeline; overload;
  function  Throttle(numEntries: integer; unblockAtCount: integer = 0):
    IOmniPipeline;
  function  WaitFor(timeout_ms: cardinal): boolean;
  property Input: IOmniBlockingCollection read GetInput;
  property Output: IOmniBlockingCollection read GetOutput;
  property PipelineStage[idxStage: integer]: IOmniPipelineStage read GetPipelineStage;
end;
```

Various `Stage` and `Stages` overloads can be used to define one or more stages and optionally configure them with a task configuration block.

The `Run` function does all the hard work. It creates queues and sets up OmniThreadLibrary tasks.

The `OnStop` function assigns a termination handler - a code that will be called when all pipeline stages stop working. This termination handler is called from one of the worker threads, not from the main thread! If you need to run a code in the main thread, use the task configuration block on the last stage.

Another possibility is to use another variation of `OnStop` that accepts a delegate with an `IOmniTask` parameter. You can then use `task.Invoke` to execute a code in the context of the main thread.

Release [3.07.2] introduced method `OnStopInvoke` which works like `OnStop` except that the termination handler is automatically executed in the context of the owner thread via implicit `Invoke`. For example, see `Parallel.ForEach.OnStopInvoke`.

The `From` function sets the input queue. It will be passed to the first stage in the `input` parameter. If your code does not call this function, OmniThreadLibrary will automatically create the input queue for you. Input queue can be accessed through the `Input` property.

The output queue is always created automatically. It can be accessed through the `Output` property. Even if the last stage doesn't produce any result, this queue can be used to signal the end of computation. [When each stage ends, CompleteAdding is automatically called on the output queue. This allows the next stage to detect the end of input (blocking collection enumerator will exit or `TryTake` will return false). Same goes on for the output queue.]

The `WaitFor` function waits for up to `timeout_ms` milliseconds for the pipeline to finish the work. It returns `True` if all stages have processed the data before the time interval expires.

The `NumTasks` function sets the number of parallel execution tasks for the stage(s) just added with the `Stage(s)` function (IOW, call `Stage` followed by `NumTasks` to do that). If it is called before

any stage is added, it will specify the default for all stages. Number of parallel execution tasks for a specific stage can then still be overridden by calling `NumTasks` after the `Stage` is called. [See the Parallel stages section below for more information.]

If `NumTasks` receives a positive parameter (`> 0`), the number of worker tasks is set to that number. For example, `NumTasks(16)` starts 16 worker tasks, even if that is more then number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies number of cores that should be reserved for other use. Number of worker task is then set to `<number of available cores> - <number of reserved cores>`. If, for example, current process can use 16 cores and `NumTasks(-4)` is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

The `Throttle` functions sets the throttling parameters for stage(s) just added with the `Stage(s)` function. Just as the `NumTask` it affects either the global defaults or just currently added stage(s). By default, throttling is set to 10,240 elements. [See the Throttling section below for more info.]

Up to version [3.07] it was not possible to fully disable the throttling mechanism. To fix this problem, version [3.07] introduced method `NoThrottle` which disables throttling on stage(s) just added with the `Stage(s)` function.

The `HandleExceptions` function changes the stage wrapper code so that it will pass exceptions from the input queue to the stage code. Just as the `NumTask` it affects either the global defaults or just currently added stage(s). [See the Exceptions section below for more info.]

The `PipelineStage[]` index property allows the program to access input and output pipeline of each stage. This allows the code to insert messages at any point in the pipeline.

```
1 IOmniPipelineStage = interface
2   property Input: IOmniBlockingCollection read GetInput;
3   property Output: IOmniBlockingCollection read GetOutput;
4 end;
```

3.10.3.1 Example

An example will help explain all this.

```
1 Parallel.Pipeline
2   .Throttle(1000)
3   .Stage(StageGenerate)
4   .Stage(StageMult2)
5   .Stages([StageMinus3, StageMod5])
6     .NumTasks(2)
7   .Stage(StageSum)
8   .Run;
```

First, a global throttling parameter is set. It will be applied to all stages. Two stages are then added, each with a separate call to the `Stage` function.

Another two stages are then added with one call. They are both set to execute in two parallel tasks. At the end another stage is added and the whole setup is executed.

The complete process will use seven tasks (one for StageGenerate, one for StageMult2, two for StageMinus3, two for StageMod5 and one for StageSum).

3.10.4 Generators, mutators, and aggregators

Let's take a look at three different examples of multiprocessing stages.

A stage may accept no input and just generate an output. This will only happen in the first stage. (A middle stage accepting no input would render the whole pipeline rather useless.)

```
1 procedure StageGenerate(const input, output: IOmniBlockingCollection);
2 var
3   i: integer;
4 begin
5   for i := 1 to 1000000 do
6     output.Add(i);
7 end;
```

A stage may also read data from the input and generate the output. Zero, one or more elements could be generated for each input.

```
1 procedure StageMult2(const input, output: IOmniBlockingCollection);
2 var
3   value: TOmniValue;
4 begin
5   for value in input do
6     output.Add(2 * value.AsInteger);
7 end;
```

The last example is a stage that reads data from the input, performs some operation on it and returns the aggregation of this data.

```
 1 procedure StageSum(const input, output: IOmniBlockingCollection);
 2 var
 3   sum : integer;
 4   value: TOmniValue;
 5 begin
 6   sum := 0;
 7   for value in input do
 8     Inc(sum, value);
 9   output.Add(sum);
10 end;
```

All examples are just special cases of the general principle. Pipeline doesn't enforce any correlation between the amount of input data and the amount of output data. There's also absolutely no requirement that data must be all numbers. Feel free to pass around anything that can be contained in a `TOmniValue`.

3.10.5 Throttling

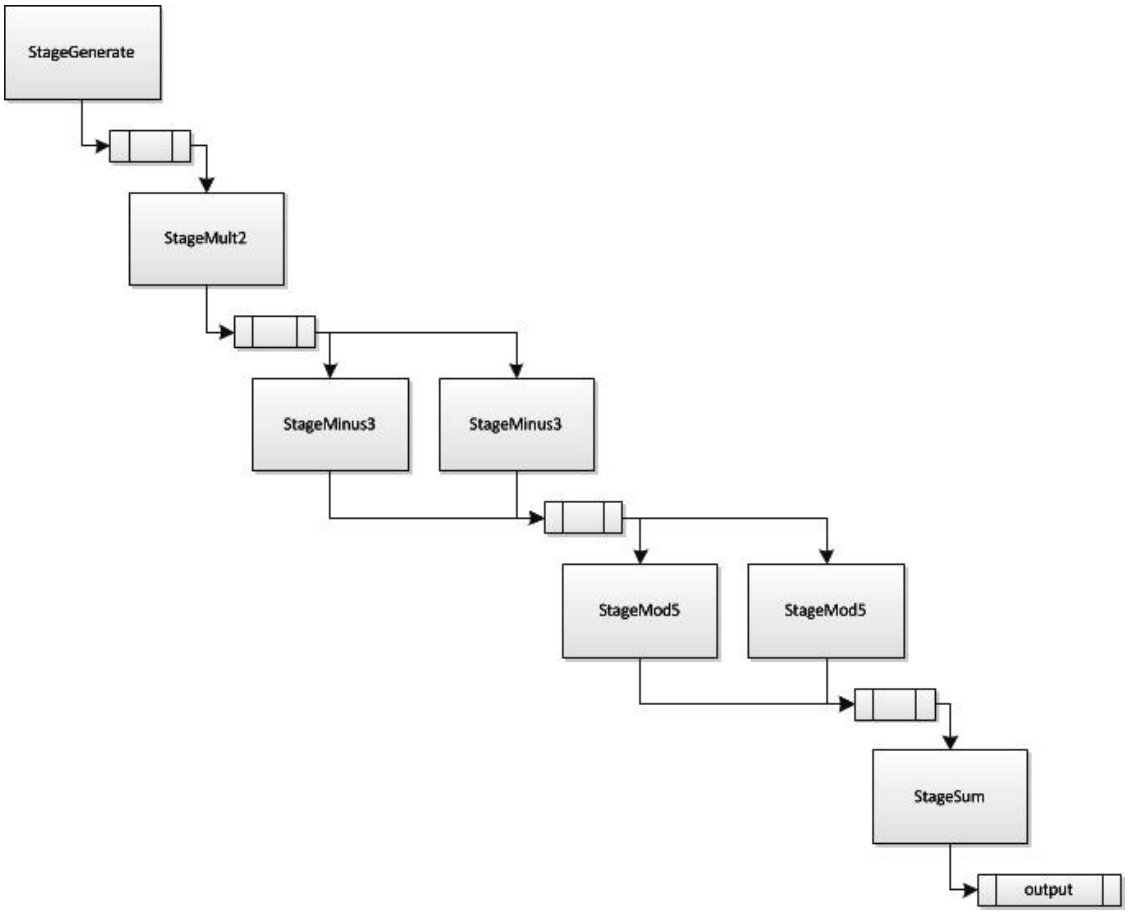
In some cases, large amount of data may be passed through the multistage process. If one stage is suspended for some time – or if it performs a calculation that is slower than the calculation in the previous stage – this stage's input queue may fill up with data which can cause lots of memory to be allocated and later released. To even out the data flow, Pipeline uses throttling.

Throttling sets the maximum size of the blocking collection (in `TOmniValue` units). When the specified quantity of data items is stored in the collection, no more data can be added. The `Add` function will simply block until the collection is empty enough or `CompleteAdding` has been called. Collection is deemed to be empty enough when the data count drops below some value which can be either passed as a second parameter to the `Throttle` function or is calculated as a 3/4 of the maximum size limit if the second parameter is not provided.

3.10.6 Parallel stages

Usually, one thread is started for each stage in the pipeline. In some specialized cases,

however, it may be desirable to run more than one task for a stage.



There's always only one queue sitting between stages even if there are multiple tasks executing a stage. This is easily accomplished by using `IOmniBlockingCollection` which supports multiple readers and multiple writers in a threadesafe manner.

There's an important caveat, though. If you split a stage into multiple tasks, data will be processed in an indeterminate order. You cannot know how many items will be processed by each task and in which order they will be processed. Even worse – data will exit multitask stage in an indeterminate order (data output from one task will be interleaved with the data from the other task). As of this moment there's no way to enforce original ordering.

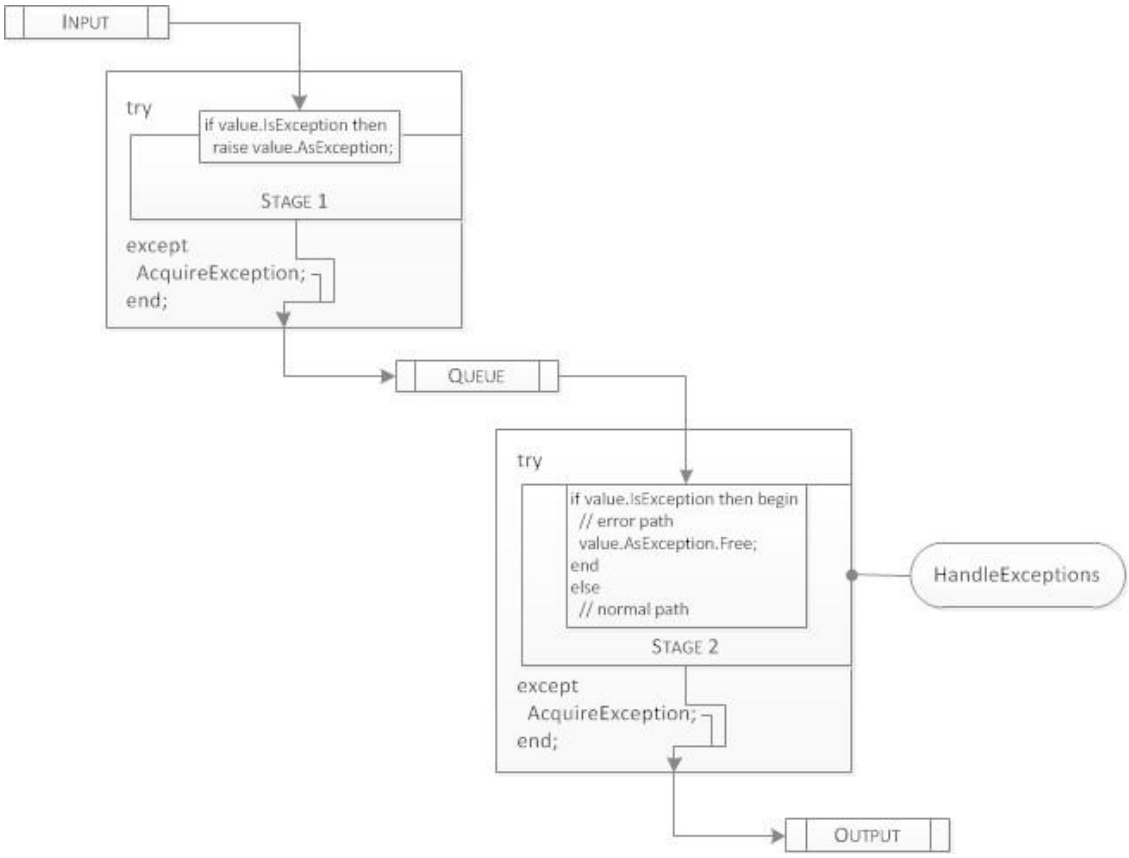
3.10.7 Exceptions

In the Pipeline abstraction, exceptions are passed through the interconnecting queues from one stage to another.

If an unhandled exception occurs in a stage, it gets caught by the wrapper code and is re-queued to the output queue. When data element containing exception is read by the next stage, it automatically generates an exception which gets passed to the next output queue. In this way, exception will progress through the pipeline and will be inserted into the output queue.

If you want to handle exceptions in one of the stages, call the `HandleExceptions` function after declaring the stage. (You can also call this function before declaring any stage - that way it will change behaviour for all stages.) You can then call the `IsException` and `AsException` functions on the input value to check whether a value contains an exception and to access this exception.

Handling the last (output) stage is slightly different. If you don't want to reraise exceptions when data is read from the pipeline output, you have to turn the reraise exception flag off on the output queue by calling `pipeline.Output.ReraiseExceptions(false)`.



The following example demonstrates the use of exception-handling functions.

```
1  procedure StageException1(const input: TOmniValue; var output: TOmniValue);
2  begin
3    output := input.AsInteger * 42;
4  end;
5
6  procedure StageException2(const input, output: IOmniBlockingCollection);
7  var
8    outVal: TOmniValue;
9    value : TOmniValue;
10 begin
11   for value in input do begin
12     if value.IsException then begin
13       value.AsException.Free;
14       outVal.Clear;
15     end
16     else
17       outVal := 1 / value.AsInteger;
18     if not output.TryAdd(outVal) then
19       break; //for
20   end;
21 end;
22
23 procedure TfrmOtlParallelExceptions.btnPipeline1Click(Sender: TObject);
24 var
25   pipeline: IOmniPipeline;
26   value   : TOmniValue;
27 begin
28   //Stage 2 should accept and correct stage 1 exception
29   //(third output will be empty)
30   pipeline := Parallel.Pipeline
31     .Stage(StageException1)
32     .Stage(StageException2)
```

```
33        .HandleExceptions
34      .Run;
35
36    // Provide input
37    with pipeline.Input do begin
38      // few normal elements
39      Add(1);
40      Add(2);
41      // then trigger the exception in the first stage;
42      // this exception should be 'corrected' in the second stage
43      Add('three');
44      Add(4);
45      CompleteAdding;
46    end;
47
48    // Process output; there should be no exception in the output collection
49    for value in pipeline.Output do
50      Log(value.AsString);
51  end;
```

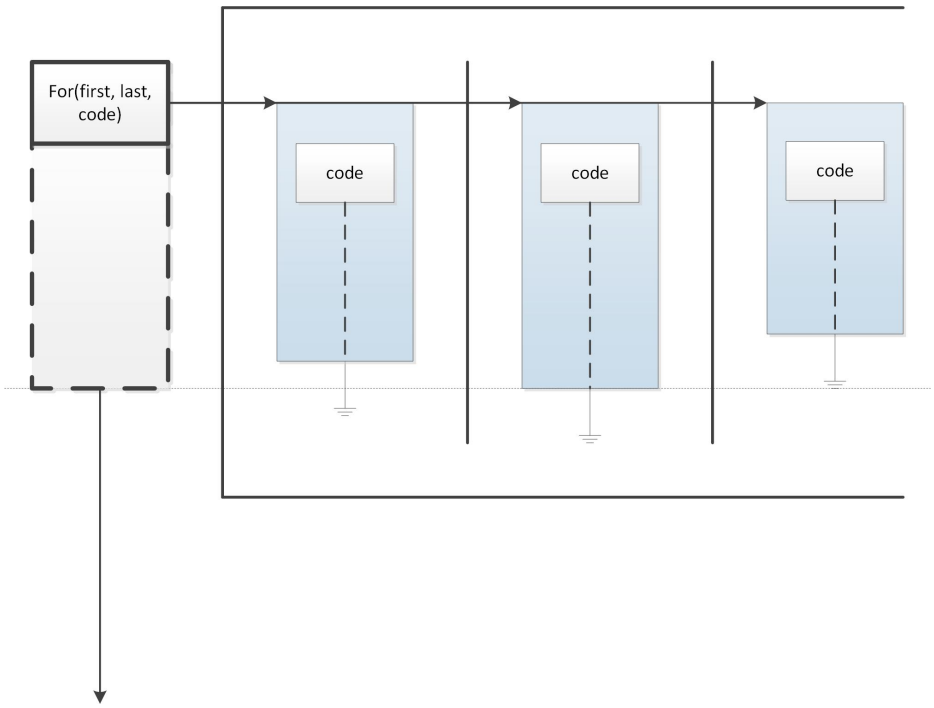
See also demo 48_OtlParallelExceptions.

3.10.8 Examples

Practical example of Pipeline usage can be found in chapter Web download and database storage.

3.11 Parallel for

Parallel for abstraction creates a parallel `for` loop that iterates over an integer range in multiple threads. To create a parallel for loop, call `Parallel.For`. Since version [3.06], Parallel for abstraction can iterate over elements of an array. This kind of parallel for loop is created by calling `Parallel.For<T>`.



`Parallel.For` provides a simple way to parallelize a `for` loop. When you use `Parallel.For`, OmniThreadLibrary starts multiple background tasks, each working on a subset of `for` loop ranges. By default, `For` waits for all background threads to complete before the control is returned to the caller.

See also demos 57_For and 58_ForVsForEach.

OmniThreadLibrary also implements ForEach abstraction, which is much more powerful than Parallel for but also quite slower.

Example:

```
1 PrimeCount.Value := 0;
2 Parallel.For(1, 1000000).Execute(
3   procedure (value: integer)
4   begin
5     if IsPrime(value) then
6       PrimeCount.Increment;
7     end;
8   end);
```

This simple program calculates number of prime numbers in the range from one to one million. The `PrimeCount` object must be capable of atomic increment (a thread-safe increment), which is simple to achieve with the use of the `TOmniAlignedInt32` record. The ForEach task is coded as an anonymous method but you can also use a normal method or a normal procedure for the task code.

3.11.1 IOmniParallelSimpleLoop interface

The `Parallel.For` function returns an `IOmniParallelSimpleLoop` interface which is used to configure and run the parallel for loop.

```
1 type
2   TOmniIteratorSimpleSimpleDelegate = reference to procedure(value: integer);
3   TOmniIteratorSimpleDelegate = reference to procedure(taskIndex, value: integer);
4   TOmniIteratorSimpleFullDelegate = reference to procedure(const task: IOmniTask;
5     taskIndex, value: integer);
6   TOmniSimpleTaskInitializerDelegate = reference to procedure(taskIndex, fromIndex,
7     toIndex: integer);
8   TOmniSimpleTaskInitializerTaskDelegate = reference to procedure(const task: IOmniTask;
9     taskIndex, fromIndex, toIndex: integer);
10  TOmniSimpleTaskFinalizerDelegate = reference to procedure(taskIndex, fromIndex,
11    toIndex: integer);
12  TOmniSimpleTaskFinalizerTaskDelegate = reference to procedure(const task: IOmniTask;
13    taskIndex, fromIndex, toIndex: integer);
14
15  IOmniParallelSimpleLoop = interface
16    function  CancelWith(const token: IOmniCancellationToken): IOmniParallelSimpleLoop;
17    function  NoWait: IOmniParallelSimpleLoop;
18    function  NumTasks(taskCount : integer): IOmniParallelSimpleLoop;
19    function  OnStop(stopCode: TProc): IOmniParallelSimpleLoop; overload;
20    function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelSimpleLoop; overload;
21    function  OnStopInvoke(stopCode: TProc): IOmniParallelSimpleLoop;
22    function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelSimpleLoop;
23    procedure Execute(loopBody: TOmniIteratorSimpleSimpleDelegate); overload;
24    procedure Execute(loopBody: TOmniIteratorSimpleDelegate); overload;
25    procedure Execute(loopBody: TOmniIteratorSimpleFullDelegate); overload;
26    function  Initialize(taskInitializer: TOmniSimpleTaskInitializerDelegate):
27      IOmniParallelSimpleLoop; overload;
28    function  Initialize(taskInitializer: TOmniSimpleTaskInitializerTaskDelegate):
29      IOmniParallelSimpleLoop; overload;
30    function  Finalize(taskFinalizer: TOmniSimpleTaskFinalizerDelegate):
31      IOmniParallelSimpleLoop; overload;
32    function  Finalize(taskFinalizer: TOmniSimpleTaskFinalizerTaskDelegate):
33      IOmniParallelSimpleLoop; overload;
34    function  WaitFor(maxWait_ms: cardinal): boolean;
35  end;
```

`Execute` accepts the block of code to be executed for each value in the input container. Three method signatures are supported.

During execution, input data is split into `N` sequential ranges (where `N` is number of

concurrently running tasks). Each tasks is assigned a task index (from 0 to `N-1`). This task index can be used for task-specific initialization and is accessible from `Initialize`, `Finalize` and `Execute` delegates.

`CancelWith` enables the cancellation mechanism.

With `Initialize` you can initialize per-task data before the task begins execution. Such data can be finalized (cleaned up) with `Finalize`.

If you call the `NoWait` function, Parallel for will start in the background and control will be returned to the main thread immediately. If `NoWait` is not called, `Execute` will only return after all tasks have stopped working.

By calling `NumTasks` you can set up the number of worker tasks. By default, number of tasks is set to [number of cores available to the process] - 1 if `NoWait` or `PreserveOrder` modifiers are used and to [number of cores available to the process] in all other cases.

If `NumTasks` receives a positive parameter (`> 0`), the number of worker tasks is set to that number. For example, `NumTasks(16)` starts 16 worker tasks, even if that is more then number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies number of cores that should be reserved for other use. Number of worker task is then set to `<number of available cores> - <number of reserved cores>`. If, for example, current process can use 16 cores and `NumTasks(-4)` is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

`OnStop` sets up a termination handler which will be called after all tasks will have completed their work. If `NoWait` function was called, `OnStop` will be called from one of the worker threads. If, however, `NoWait` function was not called, `OnStop` will be called from the thread that created the Parallel for abstraction.

Release [3.07.2] introduced method `OnStopInvoke` which works like `OnStop` except that the termination handler is automatically executed in the context of the owner thread via implicit `Invoke`. For example, see `Parallel.ForEach.OnStopInvoke`.

`TaskConfig` sets up a task configuration block. Same task configuration block will be applied to all worker tasks.

The following example uses `TaskConfig` to set up a message handler which will receive messages sent from worker tasks.

```
1 FParallel := Parallel.For(1, 17)
2   .TaskConfig(Parallel.TaskConfig.OnMessage(Self))
3   .NoWait
4   .OnStop(procedure begin FParallel := nil; end);
5
6 FParallel
7   .Execute(
8     procedure (const task: IOmniTask; taskIndex, value: integer)
9     begin
10      task.Comm.Send(WM_LOG, value);
11    end);
```

A call to the `WaitFor` function will wait for up to `timeout_ms` milliseconds (this value can be set to `INFINITE`) for all background tasks to terminate. If the tasks terminate in the specified time, `WaitFor` will return `True`. Otherwise, it will return `False`.

3.11.2 Iterating over an array

Since version [3.06] you can use `Parallel.For<T>(const arr: TArray<T>)` to iterate over elements of an array. `Parallel.For<T>` returns a `IOmniParallelSimpleLoop<T>` interface which is almost the same as the `IOmniParallelSimpleLoop` interface returned from `Parallel.For` except that it operates on elements of the array (`value: T`) and not on indexes (`value: integer`).

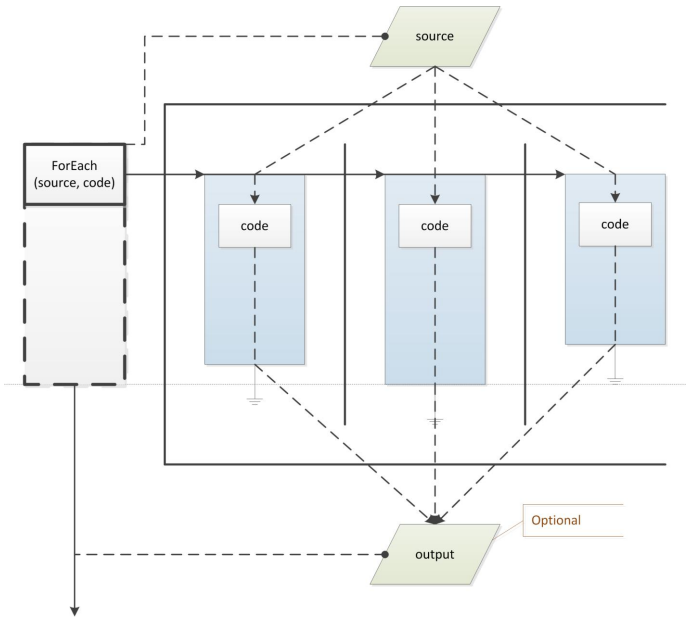
```
type
  TOmniIteratorSimpleSimpleDelegate<T> = reference to procedure(var value: T);
  TOmniIteratorSimpleDelegate<T> = reference to procedure(taskIndex: integer;
    var value: T);
  TOmniIteratorSimpleFullDelegate<T> = reference to procedure(const task: IOmniTask;
    taskIndex: integer; var value: T);

  IOmniParallelSimpleLoop<T> = interface
    function  CancelWith(const token: IOmniCancellationToken): IOmniParallelSimpleLoop<T>;
    function  NoWait: IOmniParallelSimpleLoop<T>;
    function  NumTasks(taskCount : integer): IOmniParallelSimpleLoop<T>;
    function  OnStop(stopCode: TProc): IOmniParallelSimpleLoop<T>; overload;
    function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelSimpleLoop<T>;
      overload;
    function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelSimpleLoop<T>;
    procedure Execute(loopBody: TOmniIteratorSimpleSimpleDelegate<T>); overload;
    procedure Execute(loopBody: TOmniIteratorSimpleDelegate<T>); overload;
    procedure Execute(loopBody: TOmniIteratorSimpleFullDelegate<T>); overload;
    function  Initialize(taskInitializer: TOmniSimpleTaskInitializerDelegate): IOmniParall\
elSimpleLoop<T>; overload;
    function  Initialize(taskInitializer: TOmniSimpleTaskInitializerTaskDelegate): IOmniPa\
rallelSimpleLoop<T>; overload;
    function  Finalize(taskFinalizer: TOmniSimpleTaskFinalizerDelegate): IOmniParallelSimp\
leLoop<T>; overload;
    function  Finalize(taskFinalizer: TOmniSimpleTaskFinalizerTaskDelegate): IOmniParallel\
SimpleLoop<T>; overload;
    function  WaitFor(maxWait_ms: cardinal): boolean;
  end;
```

3.11.3 Examples

Practical example of Parallel.For usage can be found in chapter Using task initializers in parallel for.

3.12 ForEach

ForEach abstraction creates a parallel `for` loop that iterates over a range of data (number range, list, queue, dataset ...) in multiple threads. To create a ForEach abstraction, call `Parallel.ForEach`.



When you use `Parallel.ForEach`, OmniThreadLibrary starts multiple background tasks and connects them to the source through a serialization mechanism. Output is optionally sorted in the order of the input. By default, `ForEach` waits for all background threads to complete before the control is returned to the caller.

See also demos `35_ParallelFor` and `36_OrderedFor`.

OmmiThreadLibrary also implements Parallel for abstraction, which is more limited than ForEach but runs faster.

Example:

```
1 PrimeCount.Value := 0;
2 Parallel.ForEach(1, 1000000).Execute(
3   procedure (const value: integer)
4   begin
5     if IsPrime(value) then
6       PrimeCount.Increment;
7     end;
8   end);
```

This simple program calculates number of prime numbers in the range from one to one million. The `PrimeCount` object must be capable of atomic increment (a thread-safe increment), which is simple to achieve with the use of the `TOmniAlignedInt32` record. The ForEach task is coded as an anonymous method but you can also use a normal method or a normal procedure for the task code.

3.12.1 Cooperation

The main point of the ForEach abstraction is cooperation between the parallel tasks. ForEach goes to great lengths to minimize possible clashes between threads when they access the source data. Except in special occasions (number range, `IOmniBlockingCollection`), source data is not thread-safe and locking must be used to synchronize access.

To minimize this locking, source data is allocated to worker tasks in blocks. ForEach creates a source provider object which accesses the source data in a thread-safe manner. This source provider makes sure to always return an appropriately-sized block of source data (size will depend on number of tasks, type of the source data and other factors) when a task runs out of data to process.

Because the source data is allocated in blocks, it is possible that one of the tasks runs out of work while other tasks are still busy. In this case, a task will steal data from one of the other tasks. This approach makes all tasks as busy as possible while minimizing the contention.

The details of this process are further discussed in section Internals below.

3.12.2 Iterating over ...

The `Parallel` class defines many `ForEach` overloads, each supporting different container type. We will look at them in more detail in the following sections.

3.12.2.1 ... Number ranges

To iterate over a range, pass `first` and `last` index to the `ForEach` call. Optionally, you can pass a `step` parameter, which defaults to `1`. `ForEach` will then iterate from `first` to `last` with a `step` increment.

```
1 class function ForEach(low, high: integer; step: integer = 1):
2   IOmniParallelLoop<integer>; overload;
```

The pseudocode for numeric `ForEach` could be written as

```
1 i := low;
2 while ((step > 0) and (i <= high)) or
3       ((step < 0) and (i >= high)) do
4 begin
5   // process 'i' in parallel
6   if low < high then Inc(i, step)
7                 else Dec(i, step);
8 end;
```

3.12.2.2 ... Enumerable collections

If you want to iterate over a collection (say, a `TStringList`), you have two possibilities.

One is to use an equivalent of `for i := 0 to sl.Count-1 do Something(sl[i])`.

```
1 Parallel.ForEach(0, sl.Count-1).Execute(
2   procedure (const value: integer)
3   begin
4     Something(sl[value]);
5   end);
```

Another is to use an equivalent of `for s in sl do Something(s)`.

```
1 Parallel.ForEach(sl).Execute(
2   procedure (const value: TOmniValue)
3   begin
4     Something(value);
5   end);
```

In the second example, value is passed to the task function as a `TOmniValue` parameter. In the example above, it will be automatically converted into a string, but sometimes you'll have to do it manually, by calling `value.AsString` (or use appropriate casting function when iterating over a different container).

A variation of the second approach is to tell the `ForEach` that the container contains strings. OmniThreadLibrary will then do the conversion for you.

```
1 Parallel.ForEach<string>(sl).Execute(
2   procedure (const value: string)
3   begin
```

```
4     Something(value);
5   end);
```

You may wonder which of those approaches is better. The answer depends on whether you can simultaneously access different items in the container from different threads at the same time. In other words, you have to know whether the container is thread-safe for reading. Luckily, all important Delphi containers (`TList`, `TObjectList`, `TStringList`) fall into this category.

If the container is thread-safe for reading, then the numeric approach (`ForEach(0, sl.Count-1)`) is **much** faster than the for..in approach (`ForEach(sl)`). The speed difference comes from the locking - in the former example `ForEach` never locks anything and in the latter example locking is used to synchronize access to the container.

However, if the container is not thread-safe for reading, you **have** to use the latter approach.

There are three ways to iterate over enumerable containers. You can provide the `ForEach` call with an `IEnumerable` interface, with an `IEnumerator` interface or with an enumerable collection itself. In the latter case, OmniThreadLibrary will use RTTI to access the enumerator for the collection. For this to work, enumerator itself must be implemented as an object, not as a record or interface. Luckily, most if not all of the VCL enumerators are implemented in this way.

```
1 class function  ForEach(const enumerable: IEnumerable):
2   IOmniParallelLoop; overload;
3 class function  ForEach(const enum: IEnumerator):
4   IOmniParallelLoop; overload;
5 class function  ForEach(const enumerable: TObject):
6   IOmniParallelLoop; overload;
7 class function  ForEach<T>(const enumerable: IEnumerable):
8   IOmniParallelLoop<T>; overload;
9 class function  ForEach<T>(const enum: IEnumerator):
10  IOmniParallelLoop<T>; overload;
11 class function  ForEach<T>(const enumerable: TEnumerable<T>):
12  IOmniParallelLoop<T>; overload;
13 class function  ForEach<T>(const enum: TEnumerator<T>):
14  IOmniParallelLoop<T>; overload;
15 class function  ForEach<T>(const enumerable: TObject):
16  IOmniParallelLoop<T>; overload;
```

3.12.2.3 ... Thread-safe enumerable collections

Collection enumeration uses locking to synchronize access to the collection enumerator, which slows down the enumeration process. In some special cases, collection may be enumerable without the locking. To enumerate over such collection, it must implement `IOmniValueEnumerable` and `IOmniValueEnumerator` interfaces, which are defined in the OtlCommon unit.

```
1 class function  ForEach(const enumerable: IOmniValueEnumerable):
2   IOmniParallelLoop; overload;
3 class function  ForEach(const enum: IOmniValueEnumerator):
4   IOmniParallelLoop; overload;
5 class function  ForEach<T>(const enumerable: IOmniValueEnumerable):
6   IOmniParallelLoop<T>; overload;
7 class function  ForEach<T>(const enum: IOmniValueEnumerator):
8   IOmniParallelLoop<T>; overload;
```

3.12.2.4 ... Blocking collections

To simplify enumerating over blocking collections, the `Parallel` class implements two `ForEach` overloads accepting a blocking collection. Internally, blocking collection is enumerated

with the IOmniValueEnumerable interface.

```
class function  ForEach(const source: IOmniBlockingCollection):
  IOmniParallelLoop; overload;
class function  ForEach<T>(const source: IOmniBlockingCollection):
  IOmniParallelLoop<T>; overload;
```

3.12.2.5 ... Anything

As a last resort, the Parallel class implements three ForEach overloads that will (with some help from the programmer) iterate over any data.

The TOmniSourceProvider way is powerful, but complicated.

```
class function  ForEach(const sourceProvider: TOmniSourceProvider):
  IOmniParallelLoop; overload;
```

You must implement a descendant of the TOmniSourceProvider class. All methods must be thread-safe. For more information about the source providers, see the Internals section, below.

```
TOmniSourceProvider = class abstract
public
  function  Count: int64; virtual; abstract;
  function  CreateDataPackage: TOmniDataPackage; virtual; abstract;
  function  GetCapabilities: TOmniSourceProviderCapabilities;
    virtual; abstract;
  function  GetPackage(dataCount: integer; package: TOmniDataPackage):
    boolean; virtual; abstract;
  function  GetPackageSizeLimit: integer; virtual; abstract;
end;
```

As this approach is not for the faint of heart, OmniThreadLibrary provides a slower but much simpler version.

```
class function  ForEach(enumerator: TEnumeratorDelegate):
  IOmniParallelLoop; overload;
class function  ForEach<T>(enumerator: TEnumeratorDelegate<T>):
  IOmniParallelLoop<T>; overload;
```

Here, you must provide a function that will return next data whenever the ForEach asks for it.

```
TEnumeratorDelegate = reference to function(var next: TOmniValue): boolean;
TEnumeratorDelegate<T> = reference to function(var next: T): boolean;
```

OmniThreadLibrary will provide the synchronisation (locking) so you can be sure this method will only be called from one thread at any time. As you may expect, this will slow things down, but parallelization may still give you a reasonable performance increase if ForEach payload is substantial (i.e. if the method you are executing in the ForEach loop takes some time to execute).

The TEnumeratorDelegate function can also be used as a generator; that is it can calculate the values that will then be processed in the parallel for loop.

3.12.3 Providing external input

Sometimes, especially when you are dealing with datasets, synchronized access to the container will not be enough. When you are dealing with database connections, datasets etc you can easily run into thread affinity problems - that is the inability of some component to work correctly if it is called from a different thread than the one that it was created in.

Always initialize database connections and datasets in the thread that will use them. You code may work without that precaution but unless you have extensively tested database components in multiple threads, you should not assume that they will work correctly unless that condition (initialization and use in the same thread) is met.

In such case, the best way is to provide the input directly from the main thread. There are few different ways to achieve that.

1. Repackage data into another collection that can be easily consumed in ForEach (`TObjectList`, `TStringList`, `TOmniBlockingCollection`).
2. Run the ForEach in `NoWait` mode, then write the data into the input queue and when you run out of data, wait for the ForEach loop to terminate. This approach is also useful when you want to push ForEach into background and provide it with data from some asynchronous event handler.

An example of the second approach will help clarify the idea.

```
1  uses
2    OtlCommon,
3    OtlCollections,
4    OtlParallel;
5
6  procedure Test;
7  var
8    i    : integer;
9    input: IOmniBlockingCollection;
10   loop : IOmniParallelLoop<integer>;
11   wait : IOmniWaitableValue;
12 begin
13   // create the container
14   input := TOmniBlockingCollection.Create;
15   // create the 'end of work' signal
16   wait := CreateWaitableValue;
17   loop := Parallel.ForEach<integer>(input);
18   // set up the termination method which will signal 'end of work'
19   loop.OnStop(
20     procedure
21     begin
22       wait.Signal;
23     end);
24   // start the parallel for loop in NoWait mode
25   loop.NoWait.Execute(
26     procedure (const value: integer)
27     begin
28       // do something with the input value
29       OutputDebugString(PChar(Format('%d', [value])));
30     end
31   );
32   // provide the data to the parallel for loop
33   for i := 1 to 1000 do
34     input.Add(i);
35   // signal to the parallel for loop that there's no more data to process
36   input.CompleteAdding;
37   // wait for the parallel for loop to stop
38   wait.WaitFor;
39   // destroy the parallel for loop
40   loop := nil;
41 end;
```

3.12.4 IOmniParallelLoop interface

The `Parallel.ForEach` returns an `IOmniParallelLoop` interface which is used to configure and run the parallel for loop.

```
IOmniParallelLoop = interface
  function  Aggregate(defaultAggregateValue: TOmniValue;
    aggregator: TOmniAggregatorDelegate): IOmniParallelAggregatorLoop;
  function  AggregateSum: IOmniParallelAggregatorLoop;
  procedure Execute(loopBody: TOmniIteratorDelegate); overload;
  procedure Execute(loopBody: TOmniIteratorTaskDelegate); overload;
  function  CancelWith(const token: IOmniCancellationToken):
    IOmniParallelLoop;
  function  Initialize(taskInitializer: TOmniTaskInitializerDelegate):
    IOmniParallelInitializedLoop;
  function  Into(const queue: IOmniBlockingCollection):
    IOmniParallelIntoLoop;  overload;
  function  NoWait: IOmniParallelLoop;
  function  NumTasks(taskCount : integer): IOmniParallelLoop;
  function  OnMessage(eventDispatcher: TObject):
    IOmniParallelLoop; overload; deprecated 'use TaskConfig';
  function  OnMessage(msgID: word; eventHandler: TOmniTaskMessageEvent):
    IOmniParallelLoop; overload; deprecated 'use TaskConfig';
  function  OnMessage(msgID: word; eventHandler: TOmniOnMessageFunction):
    IOmniParallelLoop; overload; deprecated 'use TaskConfig';
  function  OnTaskCreate(taskCreateDelegate: TOmniTaskCreateDelegate):
    IOmniParallelLoop; overload;
  function  OnTaskCreate(taskCreateDelegate:
    TOmniTaskControlCreateDelegate): IOmniParallelLoop; overload;
  function  OnStop(stopCode: TProc): IOmniParallelLoop;
  function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelLoop; overload;
  function  OnStopInvoke(stopCode: TProc): IOmniParallelLoop;
  function  PreserveOrder: IOmniParallelLoop;
  function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelLoop;
end;
```

`ForEach<T>` returns an `IOmniParallelLoop<T>` interface, which is exactly the same as the `IOmniParallelLoop` except that each method returns the appropriate `<T>` version of the interface.

`Aggregate` and `AggregateSum` are used to implement aggregation. See the Aggregation section, below.

`Execute` accepts the block of code to be executed for each value in the input container. Two method signatures are supported, both having the `<T>` variant. One accepts only the iteration value parameter and another accepts an additional `IOmniTask` parameter.

```
TOmniIteratorDelegate = reference to procedure(const value: TOmniValue);
TOmniIteratorDelegate<T> = reference to procedure(const value: T);
TOmniIteratorTaskDelegate =
  reference to procedure(const task: IOmniTask; const value: TOmniValue);
TOmniIteratorTaskDelegate<T> =
  reference to procedure(const task: IOmniTask; const value: T);
```

`CancelWith` enables the cancellation mechanism.

With `Initialize` and `OnTaskCreate` you can initialize per-task data before the task begins execution. See the Task initialization section, below.

`Into` sets up the output queue, see Preserving output order.

If you call the `NoWait` function, parallel for will start in the background and control will be returned to the main thread immediately. If `NoWait` is not called, `Execute` will only return after all tasks have stopped working.

By calling `NumTasks` you can set up the number of worker tasks. By default, number of tasks is set to [number of cores available to the process] - 1 if `NoWait` or `PreserveOrder` modifiers are used and to [number of cores available to the process] in all other cases.

If NumTasks receives a positive parameter (> 0), the number of worker tasks is set to that number. For example, NumTasks(16) starts 16 worker tasks, even if that is more then number of available cores.

If NumTasks receives a negative parameter (< 0), it specifies number of cores that should be reserved for other use. Number of worker task is then set to <number of available cores> - <number of reserved cores>. If, for example, current process can use 16 cores and NumTasks(-4) is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

OnMessage functions are deprecated, use TaskConfig instead.

OnStop sets up a termination handler which will be called after all parallel for tasks will have completed their work. If NoWait function was called, OnStop will be called from one of the worker threads. If, however, NoWait function was not called, OnStop will be called from the thread that created the ForEach abstraction. This behaviour makes it hard to execute VCL code from the OnStop so release [3.02] introduced another variation accepting a delegate with an IOmniTask parameter.

```
1 TOmniTaskStopDelegate = reference to procedure (const task: IOmniTask);
2 IOmniParallelLoop = interface
3   function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelLoop;
4     overload;
5 end;
```

Using this version of OnStop, the termination handler can use task.Invoke to execute some code in the main thread. This, however, requires the ForEach abstraction to stay alive until the Invoked code is executed so you must store the result of the ForEach method in a global variable (form field, for example) and destroy it only in the termination handler.

```
1 var
2   loop: IOmniParallelLoop<integer>;
3
4 loop := Parallel.ForEach(1, N).NoWait;
5 loop.OnStop(
6   procedure (const task: IOmniTask)
7   begin
8     task.Invoke(
9       procedure
10      begin
11        // do anything
12        loop := nil;
13      end);
14  end);
15 loop.Execute(
16  procedure (const value: integer)
17  begin
18    ...
19  end);
```

Release [3.07.2] introduced method OnStopInvoke which works like OnStop except that the termination handler is automatically executed in the context of the owner thread via implicit Invoke.

The code fragment above can be rewritten using OnStopInvoke as follows.

```
1 var
2   loop: IOmniParallelLoop<integer>;
3
4 loop := Parallel.ForEach(1, N).NoWait;
5 loop.OnStopInvoke(
```

```
  procedure
  begin
    // do anything
    loop := nil;
  end);
loop.Execute(
  procedure (const value: integer)
  begin
    ...
  end);
```

`PreserveOrder` modifies the Parallel for behaviour so that output values are generated in the order of the corresponding input values. See the Preserving output order section, below.

`TaskConfig` sets up a task configuration block. Same task configuration block will be applied to all worker tasks.

The following example uses `TaskConfig` to set up a message handler which will receive messages sent from `ForEach` worker tasks.

```
FParallel := Parallel.ForEach(1, 17)
  .TaskConfig(Parallel.TaskConfig.OnMessage(Self))
  .NoWait
  .OnStop(procedure begin FParallel := nil; end);

FParallel
  .Execute(
    procedure (const task: IOmniTask; const value: integer)
    begin
      task.Comm.Send(WM_LOG, value);
    end);
```

Messages sent from the worker task are received and dispatched by the `IOmniParallelLoop` interface. This requires the ForEach abstraction to stay alive until the messages are processed so you must store the result of the `ForEach` method in a global variable (form field, for example) and destroy it only in the `OnStop` handler.

Some functions return a different interface. Typically, it only implements the `Execute` function accepting a different parameter than the 'normal' `Execute`. For example, `Aggregate` returns the `IOmniParallelAggregatorLoop` interface.

```
TOmniIteratorIntoDelegate =
  reference to procedure(const value: TOmniValue; var result: TOmniValue);

IOmniParallelAggregatorLoop = interface
  function  Execute(loopBody: TOmniIteratorIntoDelegate): TOmniValue;
end;
```

These variants of the `IOmniParallelLoop` interface will be described in following sections.

3.12.5 Preserving output order

When you run a ForEach loop, you can't tell in advance in which order elements from the input collection will be processed in. For example, the code below will generate all primes from `1` to `CMaxPrime` and write them into the output queue (`primeQueue`) in a nondeterministic order.

```
primeQueue := TOmniBlockingCollection.Create;
Parallel.ForEach(1, CMaxPrime).Execute(
  procedure (const value: integer)
  begin
    if IsPrime(value) then begin
      primeQueue.Add(value);
```

```
    end;
  end);
```

Sometimes this will represent a big problem and you'll have to write a sorting function that will resort the output before it can be processed further. To alleviate the problem, `IOmniParallelLoop` implements the `PreserveOrder` modifier. When used, ForEach internally sorts the results produced in the worker task method.

Using `PreserveOrder` also forces you to use the `Into` method which returns the `IOmniParallelIntoLoop` interface. (As you may expect, there's also the `<T>` version of that interface.)

```
TOmniIteratorIntoDelegate =
  reference to procedure(const value: TOmniValue; var result: TOmniValue);
TOmniIteratorIntoTaskDelegate =
  reference to procedure(const task: IOmniTask; const value: TOmniValue;
                         var result: TOmniValue);

IOmniParallelIntoLoop = interface
  procedure Execute(loopBody: TOmniIteratorIntoDelegate); overload;
  procedure Execute(loopBody: TOmniIteratorIntoTaskDelegate); overload;
end;
```

As you can see, the `Execute` method in `IOmniParallelIntoLoop` takes a different parameter than the 'normal' `Execute`. Because of that, you'll have to change a code that is passed to the `Execute` to return a result.

```
primeQueue := TOmniBlockingCollection.Create;
Parallel.ForEach(1, CMaxPrime)
  .PreserveOrder
  .Into(primeQueue)
  .Execute(
    procedure (const value: integer; var res: TOmniValue)
    begin
      if IsPrime(value) then
        res := value;
    end);
```

When using `PreserveOrder` and `Into`, `ForEach` calls your worker code for each input value. If the worker code sets output parameter (`res`) to any value, it will be inserted into a temporary buffer. Then the magic happens (see the Internals section, below) and as soon as the appropriate (sorted) value is available in the temporary buffer, it is inserted into the output queue (the one passed to the `Into` parameter).

You can also use `Into` without the `PreserveOrder`. This will give you queue management but no ordering.

3.12.6 Aggregation

Aggregation allows you to collect data from ForEach tasks and calculate one number that is returned to the user.

Let's start with an example - intentionally a very bad one! The following code fragment tries to calculate the number of prime numbers between `1` and `CMaxPrime`.

```
numPrimes := 0;
Parallel.ForEach(1, CMaxPrime).Execute(
  procedure (const value: integer)
  begin
    if IsPrime(value) then
      Inc(numPrimes);
  end);
```

Let's say it out loud - this code is **wrong**! Access to the shared variable is not synchronized between threads and that will make the result indeterminable. One way to solve the problem is to wrap the `Inc(numPrimes)` with locking and another is to use `InterlockedIncrement` instead of `Inc`, but both will slow down the execution a lot.

A solution to this problem is to use the `Aggregate` function.

```
1  procedure SumPrimes(var aggregate: TOmniValue; const value: TOmniValue)
2  begin
3    aggregate := aggregate.AsInt64 + value.AsInt64;
4  end;
5
6  procedure CheckPrime(const value: integer; var result: TOmniValue)
7  begin
8    if IsPrime(value) then
9      Result := 1;
10 end;
11
12 numPrimes :=
13   Parallel.ForEach(1, CMaxPrime)
14   .Aggregate(0, SumPrimes)
15   .Execute(CheckPrime);
```

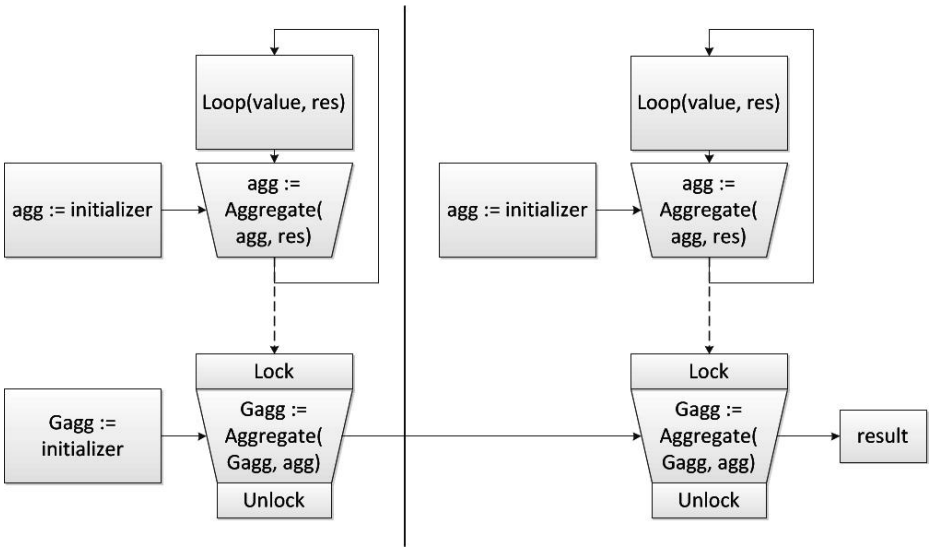
`Aggregate` takes two parameters - the first is the initial value for the aggregate and the second is an aggregation function - a piece of code that will take the current aggregate value and update it with the value returned from the worker task.

When using `Aggregate`, worker task (the code passed to the `Execute` function) has the same signature as when used with `Into`. It takes the current iteration value and optionally produces a result.

We could approximate the code above with the following `for` loop which works the same, but uses only one thread.

```
1  agg := 0;
2  result.Clear;
3  for value := 1 to CMaxPrime do begin
4    CheckPrime(value, result);
5    if not result.IsEmpty then begin
6      SumPrimes(agg, result);
7      result.Clear;
8    end;
9  end;
10 numPrimes := agg;
```

`ForEach` executes the aggregation in two stages. While the worker task is running, it will aggregate data into a local variable. When it runs out of work, it will call the same aggregation method to aggregate this local variable into a global result. In this second stage, however, locking will be used to protect the access to the global result.



Because summation is the most common usage of aggregation, `IOmniParallelLoop` implements function `AggregateSum`, which works exactly the same as the `SumPrimes` above.

```
numPrimes :=
  Parallel.ForEach(1, CMaxPrime)
  .AggregateSum
  .Execute(
    procedure (const value: integer; var result: TOmniValue)
    begin
      if IsPrime(value) then
        Result := 1;
    end
  );
```

Aggregation function can do something else but the summation. The following code segment uses aggregation to find the length of the longest line in a file.

```
function GetLongestLineInFile(const fileName: string): integer;
var
  maxLength: TOmniValue;
  sl       : TStringList;
begin
  sl := TStringList.Create;
  try
    sl.LoadFromFile(fileName);
    maxLength := Parallel.ForEach<string>(sl)
      .Aggregate(0,
        procedure(var aggregate: TOmniValue; const value: TOmniValue)
        begin
          if value.AsInteger > aggregate.AsInteger then
            aggregate := value.AsInteger;
        end)
      .Execute(
        procedure(const value: string; var result: TOmniValue)
        begin
          result := Length(value);
        end);
       Result := maxLength;
  finally FreeAndNil(sl); end;
end;
```

3.12.7 Cancellation

ForEach has a built-in cancellation mechanism. To use it, create a cancellation token and pass it to the `CancelWith` function. When a cancellation token gets signalled, all worker loops will complete the current iteration and then stop.

An example of using cancellation token can be found in the chapter Parallel search in a tree.

3.12.8 Task initialization and finalization

ForEach implements mechanism that can be used by worker tasks to intialize and destroy task-specific structures. In such cases, you have to call the `Initialize` function.

```
TOmniTaskInitializerDelegate =
  reference to procedure(var taskState: TOmniValue);
TOmniTaskFinalizerDelegate =
  reference to procedure(const taskState: TOmniValue);
TOmniIteratorStateDelegate =
  reference to procedure(const value: TOmniValue; var taskState: TOmniValue);

IOmniParallelInitializedLoop = interface
  function  Finalize(taskFinalizer: TOmniTaskFinalizerDelegate):
    IOmniParallelInitializedLoop;
  procedure Execute(loopBody: TOmniIteratorStateDelegate);
end;

IOmniParallelLoop = interface
  ...
  function  Initialize(taskInitializer: TOmniTaskInitializerDelegate):
    IOmniParallelInitializedLoop;
end;
```

You provide `Initialize` with task initializer, a procedure that will be called in each worker task when it is created and before it starts enumerating values. This procedure can initialize the `taskState` parameter with any value.

`Initialize` returns an `IOmniParallelInitializedLoop` interface which implements two functions - `Finalize` and `Execute`. Call `Finalize` to set up task finalizer, a procedure that gets called after all values have been enumerated and before the worker task ends its job.

`Execute` accepts a worker method with two parameters - the first one is the usual value from the enumerated container and the second contains the shared task state.

Of course, all those functions and interfaces are implemented in the `<T>` version, too.

The following example shows how to calculate the number of primes from `1` to `CHighPrime` by using initializers and finalizers.

```
var
  lockNum  : TOmniCS;
  numPrimes: integer;
begin
  numPrimes := 0;
  Parallel.ForEach(1, CHighPrime)
    .Initialize(
      procedure (var taskState: TOmniValue)
      begin
        taskState.AsInteger := 0;
      end)
    .Finalize(
      procedure (const taskState: TOmniValue)
      begin
        lockNum.Acquire;
        try
          numPrimes := numPrimes + taskState.AsInteger;
        finally lockNum.Release; end;
      end)
    .Execute(
      procedure (const value: integer; var taskState: TOmniValue)
```

```
22        begin
23          if IsPrime(value) then
24            taskState.AsInteger := taskState.AsInteger + 1;
25        end
26      );
27  end;
```

3.12.9 Handling exceptions

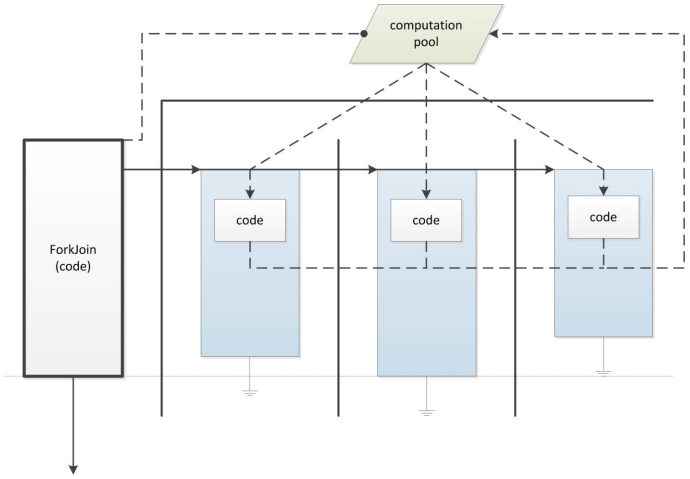
ForEach abstraction does not yet implement any exception handling. You should always wrap task method (code passed to the `Execute`) in `try..except` if you expect the code to raise exceptions.

3.12.10 Examples

Practical example of ForEach usage can be found in chapters Parallel for with synchronized output and Parallel search in a tree.

3.13 Fork/Join

Fork/Join abstraction creates a framework for solving divide and conquer algorithms.



Parallel.ForkJoin creates a computation pool into which you can submit requests (subtasks). Each subtask can create multiple sub-subtasks which are again submitted into the computation pool.

See also demos `44_Fork-Join Quicksort` and `45_Fork-Join max`.

A typical fork/join usage pattern is:

- Execute multiple subtasks.
- Wait for subtasks to terminate.
- Collect subtask results.
- Use results to compute higher-level result.

The trick here is that subtasks may spawn new subtasks and so on ad infinitum (probably a little less, or you' ll run out of stack ;)). For optimum execution, Fork/Join must therefore guarantee that the code is never running on too many background threads (an optimal value is usually equal to the number of cores in the system) and that those threads don' t run out of work.

To achieve this, ForkJoin creates multiple worker threads and connects them to a computation pool. Computation requests (i.e. subtasks) are added to this pool. They are

removed by worker threads, processed and optional new subtasks are added back to the pool.

Programs using ForkJoin typically use lots of stack space so it is advised to increase Maximum Stack Size setting in project options.

3.13.1 IOmniForkJoin interface

Fork/Join computation pool is implemented by the `IOmniForkJoin` interface and is created in the `Parallel.ForkJoin` factory function. There are two `ForkJoin` overloads – one is used for computations that don't return a result and another is used for computations that return a result of some type `T`.

```
class function ForkJoin: IOmniForkJoin; overload;
class function ForkJoin<T>: IOmniForkJoin<T>; overload;
```

Both interfaces declare just few methods.

```
IOmniForkJoin = interface
  function  Compute(action: TOmniForkJoinDelegate): IOmniCompute;
  function  NumTasks(numTasks: integer): IOmniForkJoin;
  function  TaskConfig(const config: IOmniTaskConfig): IOmniForkJoin;
end;

IOmniForkJoin<T> = interface
  function  Compute(action: TOmniForkJoinDelegate<T>): IOmniCompute<T>;
  function  NumTasks(numTasks: integer): IOmniForkJoin<T>;
  function  TaskConfig(const config: IOmniTaskConfig): IOmniForkJoin<T>;
end;
```

`Compute` creates new subtask which will execute the `action`. It returns control interface `IOmniCompute` (or `IOmniCompute<T>`).

By calling `NumTasks`, you can set the degree of parallelism. By default, fork/join uses as many threads as there are cores accessible by the process.

If `NumTasks` receives a positive parameter (`> 0`), the number of worker tasks is set to that number. For example, `NumTasks(16)` starts 16 worker tasks, even if that is more then number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies number of cores that should be reserved for other use. Number of worker task is then set to `<number of available cores> - <number of reserved cores>`. If, for example, current process can use 16 cores and `NumTasks(-4)` is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

`TaskConfig` method is used to set up a task configuration block, which is applied to each worker task.

3.13.2 IOmniCompute interface

The `IOmniCompute` interface provides interaction with the computation unit that doesn't return a result.

```
IOmniCompute = interface
  procedure Execute;
  function  IsDone: boolean;
  procedure Await;
end;
```

`Execute` executes the action that was provided to the `Compute` method. This method is used internally and should not be called from the user code.

`IsDone` checks whether the computation unit has completed the work.

`Await` waits for the computation unit to complete the work.

3.13.3 IOmniCompute<T> interface

The `IOmniCompute<T>` interface provides interaction with the computation unit that returns a result.

```
1 IOmniCompute<T> = interface
2   procedure Execute;
3   function  IsDone: boolean;
4   function  TryValue(timeout_ms: cardinal; var value: T): boolean;
5   function  Value: T;
6 end;
```

`Execute` executes the action that was provided to the `Compute` method. This method is used internally and should not be called from the user code.

`IsDone` checks whether the computation unit has completed the work.

`TryValue` waits for up to `timeout_ms` milliseconds (or as long as needed if `INFINITE` is passed for this parameter) for the computation to complete and returns the result (if available) in the `value` parameter. The function returns `True` if result is already known, `False` otherwise.

`Value` returns the computation unit result. This function will block until the result is available.

3.13.4 Exceptions

There’ s no special exception handling built into the Fork/Join abstraction at the moment. You should always catch and handle exceptions inside the action passed to the `Compute` method.

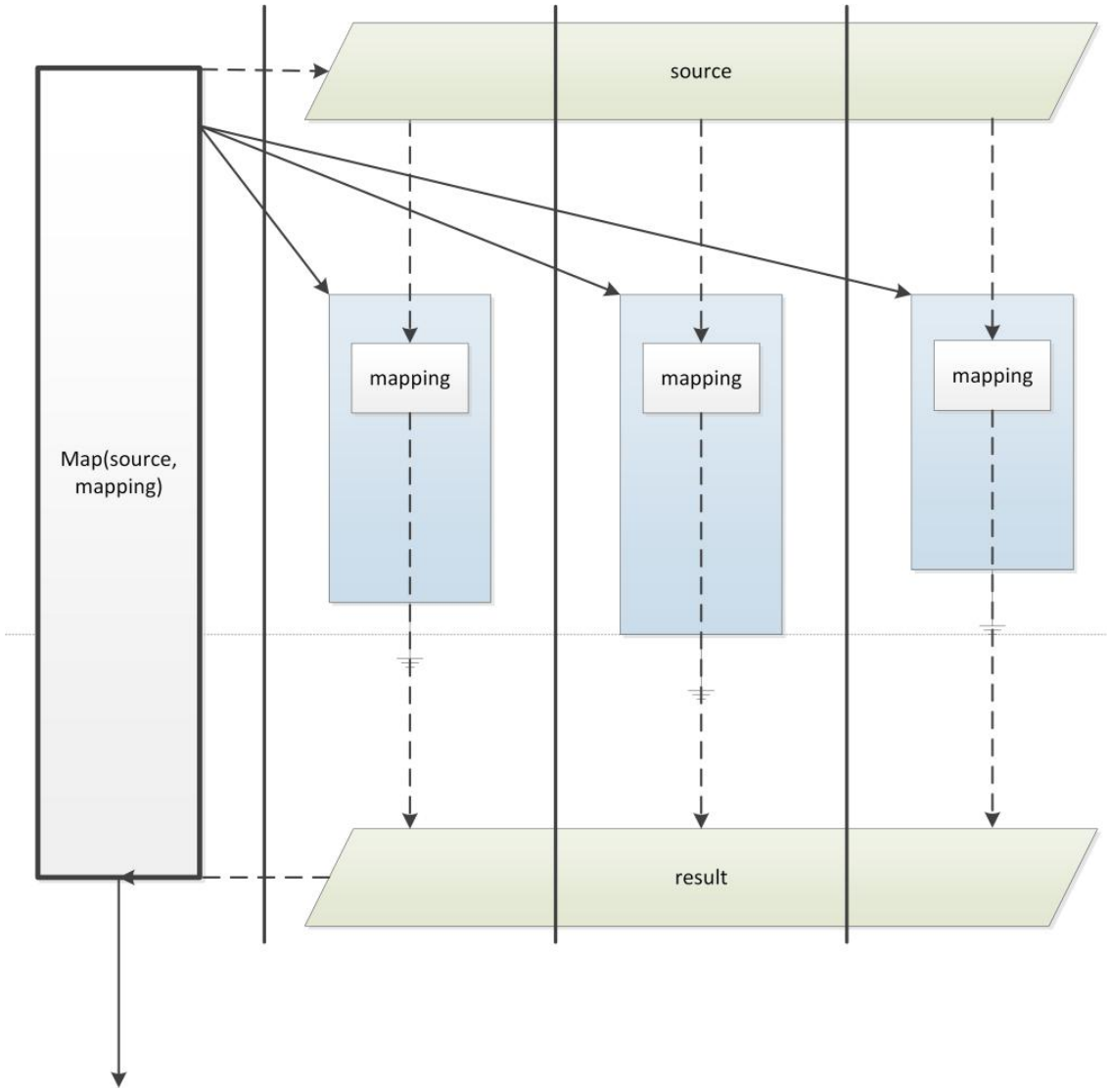
3.13.5 Examples

Practical examples of Fork/Join usage can be found in chapter QuickSort and parallel max.

3.14 Map

Map abstraction creates a parallellized mapping function which iterates over a source data array. To create a Map abstraction, call `Parallel.Map`.

`Parallel.Map` is only available on Delphi XE and newer.



When you call `Parallel.Map`, a same background task is started in multiplle blackground threads (indicated by the bold vertical line). In each thread, this task runs some filtering function (provided by the programmer) on some subset of the input array and creates output array containing data returned from the filtering function. At the end, all output arrays from background workers are combined together into the final result.

Results are ordered in the same way as corresponding input elements. (In other words – results of `Parallel.Map` are the same as results of a simple single-threaded `for` loop executing the same filtering function over the same input data.

See also demos `60_Map`.

Example:

```
1 odds := Parallel.Map<integer,string>(numbers,
2   function (const source: integer; var dest: string): boolean
3   begin
4     Result := Odd(source);
5     if Result then
6       dest := IntTostr(source);
7   end);
```

This simple code fragment takes an array of integers `numbers: TArray<integer>`, converts all odd values in that array to string representation and assigns resulting `TArray<string>` to `odds`. Worker code runs in parallel on all available cores. The filtering function is coded as an

anonymous method but you can also use a method or a normal procedure.

The `Parallel` class defines two `Map` overloads. The first creates an `IOmniParallelMapper<T1,T2>` interface which you can then further configure before workers are started.

```
type
  TMapProc<T1,T2> = reference to function(const source: T1; var target: T2): boolean;

  Parallel = class
    class function Map<T1,T2>: IOmniParallelMapper<T1,T2>; overload;
    class function Map<T1,T2>(const source: TArray<T1>;
      mapper: TMapProc<T1,T2>): TArray<T2>; overload;
    ...
  end;
```

The second overload is just a shorthand which runs mapping function on all cores and waits for the result. It is defined in the OtlParallel as foollows.

```
class function Parallel.Map<T1, T2>(const source: TArray<T1>; mapper: TMapProc<T1,T2>):
  TArray<T2>;
var
  map: IOmniParallelMapper<T1,T2>;
begin
  map := Parallel.Map<T1,T2>.Source(source);
  map.Execute(mapper);
  map.WaitFor(INFINITE);
  Result := map.Result;
end;
```

3.14.1 IOmniParallelMapper<T1,T2> interface

The `IOmniParallelMapper<T1,T2>` interface provides methods that configure and control the parallel mapper.

```
IOmniParallelMapper<T1,T2> = interface
  function  Execute(mapper: TMapProc<T1,T2>): IOmniParallelMapper<T1,T2>;
  function  NoWait: IOmniParallelMapper<T1,T2>;
  function  NumTasks(numTasks: integer): IOmniParallelMapper<T1,T2>;
  function  OnStop(stopCode: TProc): IOmniParallelMapper<T1,T2>; overload;
  function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelMapper<T1,T2>; overload;
  function  OnStopInvoke(stopCode: TProc): IOmniParallelMapper<T1,T2>;
  function  Result: TArray<T2>;
  function  Source(const data: TArray<T1>;
    makeCopy: boolean = false): IOmniParallelMapper<T1,T2>;
  function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelMapper<T1,T2>;
  function  WaitFor(maxWait_ms: cardinal): boolean;
end;
```

`Source` sets the input data array. If `makeCopy` parameter is set, data is copied to an internal array. If not (default), original data is referenced from all worker threads.

`TaskConfig` sets up a task configuration block. Same task configuration block will be applied to all worker tasks.

By calling `NumTasks` you can set up the number of worker tasks. By default, the number of tasks is set to [number of cores available to the process] - 1 if `NoWait` modifier is used and to [number of cores available to the process] if `NoWait` is not used.

If `NumTasks` receives a positive parameter (`> 0`), the number of worker tasks is set to that number. For example, `NumTasks(16)` starts 16 worker tasks, even if that is more then number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies number of cores that should be

reserved for other use. Number of worker task is then set to `<number of available cores> - <number of reserved cores>`. If, for example, current process can use 16 cores and `NumTasks(-4)` is used, only 12 (16-4) worker tasks will be started.

Value 0 is not allowed and results in an exception.

`Execute` starts the worker tasks. As a parameter it takes a mapping function which is executed in worker tasks. The same mapping function is executed from multiple tasks and must therefore be thread-safe.

If you call the `NoWait` function, Map starts in the background and control is returned to the main thread immediately. If `NoWait` is not called, `Execute` only returns after all tasks have stopped working.

`OnStop` sets up a termination handler which will be called after all parallel tasks will have completed their work. If `NoWait` function was called, `OnStop` will be called from one of the worker threads. If, however, `NoWait` function was not called, `OnStop` will be called from the thread that created the `Map` abstraction.

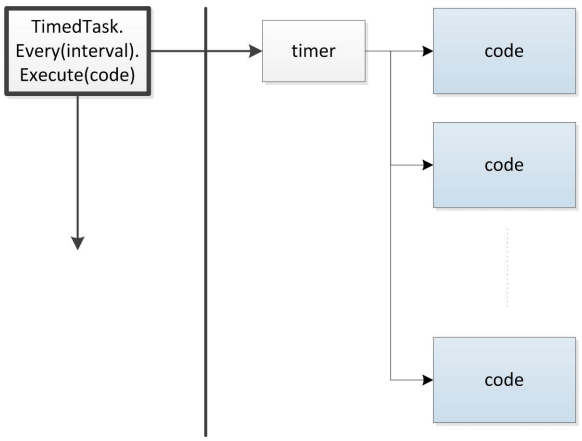
Release [3.07.2] introduced method `OnStopInvoke` which works like `OnStop` except that the termination handler is automatically executed in the context of the owner thread via implicit `Invoke`. For example, see `Parallel.ForEach.OnStopInvoke`.

A call to the `WaitFor` function will wait for up to `timeout_ms` milliseconds (this value can be set to `INFINITE`) for all background tasks to terminate. If the tasks terminate in the specified time, `WaitFor` returns `True`. Otherwise, it returns `False`.

The `Result` function returns the resulting array. This function should only be called after all worker threads have finished their work (i.e. from `OnStop`, after successfull `WaitFor`, or after `Execute` if `NoWait` is not used).

3.15 Timed task

Timed task abstraction creates a threaded timer. To create a Timed task, call `Parallel.TimedTask`.



When you call `Parallel.TimedTask`, a background task is started in a background thread. At a specified interval it will execute an anonymous method. You can think of it as of a `TTimer` which executes the `OnTimer` event in a background thread.

See also demos `65_TimedTask`.

Example:

```
FTimedTask := Parallel.TimedTask.Every(5000).Execute(
  procedure (const task: IOmniTask)
  begin
    task.Comm.Send(MSG_WATCHDOG, 'Still alive ...');
  end;
```

This code fragment creates a Timed task which every 5 seconds (5000 milliseconds) sends a message `MSG_WATCHDOG` to its owner.

3.15.1 IOmniTimedTask interface

The `Parallel.TimedTask` function returns an `IOmniTimedTask` interface which is used to configure and control a Timed task.

```
type
  TOmniTaskDelegate = reference to procedure(const task: IOmniTask);

  IOmniTimedTask = interface
    function  Every(interval_ms: integer): IOmniTimedTask;
    function  Execute(const aTask: TProc): IOmniTimedTask; overload;
    function  Execute(const aTask: TOmniTaskDelegate): IOmniTimedTask; overload;
    procedure ExecuteNow;
    procedure Start;
    procedure Stop;
    function  TaskConfig(const config: IOmniTaskConfig): IOmniTimedTask;
    function  Terminate(maxWait_ms: cardinal): boolean;
    function  WaitFor(maxWait_ms: cardinal): boolean;
    property Active: boolean;
    property Interval: integer;
  end;
```

`Every` sets the timer interval (in milliseconds). Interval must be a positive number. If you set it to a value which is less than or equal to zero, timer will be disabled. (It is, however, advisable to use `Stop` or `Active := false` for this purpose as that more clearly states the intention of the code.)

When a timed task is created, its interval is set to 0 ms.

You can inspect and modify the interval by using the `Interval` property. Setting an interval by assigning a value to this property (`Interval := value;`) is equivalent to calling `Every(value)`.

`Execute` specifies the code that will be called each `Interval` milliseconds (timer event handler code). This can be a parameterless anonymous method (or normal procedure, or an object method) or procedure/method/anonymous method accepting one parameter of type `IOmniTask` which holds the interface of the background task executing the timer method.

After you set an interval (even if the current value is equal to the previous value), the background task will wait for `Interval` milliseconds before calling the timer event handler code.

If you want to execute timer event handler immediately, call the `ExecuteNow` method. This will also reset the timer so that next automatic invocation of the timer event handler will occur `Interval` millisconds from now.

`Start` will start (enable) the timer. To stop (disable) the timer call the `Stop` method. Current state of the timer is available via the `Active` property.

You can also start/stop a timer by changing the `Active` property. Setting `Active := true` is equivalent to calling `Start` and setting `Active := false` is equivalent to calling `Stop`.

A timer is automatically started (enabled) when you call `Execute` – if and only if the `Interval` has already been set.

If you want to create a stopped timer with a very small `Interval` value, it is advisable to use the following pattern:

```
1 timedTask := Parallel.TimedTask.Execute(SomeCode);
2 timedTask.Stop;
3 timedTask.Interval := 1;
```

`TaskConfig` sets up a task configuration block.

Calling `Terminate` will stop the timed task. If it stops in `maxWait_ms`, `True` is returned, `False` otherwise. `WaitFor` waits for the task to stop (without commanding it to stop beforehand so you would have to call `Terminate` before `WaitFor`) and returns `True`/`False` just as `Terminate` does.

It is usually enough to just set the `IOmniTimedTask` instance to `nil` as that effectively calls the `Terminate(INFINITE)`.

4. Low-level multithreading

The low-level OmniThreadLibrary layer focuses on the task concept. In most aspects this is similar to the Delphi's TThread approach except that OmniThreadLibrary focuses on the code (a.k.a. task) and interaction with the code while the Delphi focuses on the operating system primitive required for executing additional threads.

A task is created using the CreateTask function, which takes as a parameter a global procedure, a method, an instance of the TOmniWorker class (or, usually, a descendant of that class) or an anonymous procedure (in Delphi 2009 and newer). CreateTask will also accept an optional second parameter, a task name, which will be displayed in the Delphi's Thread view on the thread running the task.

```
type
  TOmniTaskProcedure = procedure(const task: IOmniTask);
  TOmniTaskMethod = procedure(const task: IOmniTask) of object;
  TOmniTaskDelegate = reference to procedure(const task: IOmniTask);

function CreateTask(worker: TOmniTaskProcedure; const taskName: string = ''):
  IOmniTaskControl; overload;
function CreateTask(worker: TOmniTaskMethod; const taskName: string = ''):
  IOmniTaskControl; overload;
function CreateTask(worker: TOmniTaskDelegate; const taskName: string = ''):
  IOmniTaskControl; overload;
function CreateTask(const worker: IOmniWorker; const taskName: string = ''):
  IOmniTaskControl; overload;
```

CreateTask returns a feature-full interface IOmniTaskControl which we will explore in this chapter. The most important function in this interface, Run, creates a new thread and starts your task in it.

4.1 Low-level for the impatient

The following code represents the simplest possible low-level OmniThreadLibrary example. It executes the Beep function in a background thread. The Beep function merely beeps and exits. By exiting from the task function, the Windows thread running the task is also terminated.

```
procedure TfrmTestSimple.Beep(const task: IOmniTask);
begin
  //Executed in a background thread
  MessageBeep(MB_ICONEXCLAMATION);
end;

CreateTask(Beep, 'Beep').Run;
```

Another way to start a task is to call a Schedule function which starts it in a thread allocated from a thread pool. This is covered in the Thread pooling chapter.

4.2 Four ways to create a task

Let's examine all four ways of creating a task. The simplest possible way (demoed in application 2_TwoWayHello) is to pass a name of a global procedure to the CreateTask. This global procedure must accept one parameter of type IOmniTask .

```
procedure RunHelloWorld(const task: IOmniTask);
begin
  //
end;

CreateTask(RunHelloWorld, 'HelloWorld').Run;
```

A variation on the theme is passing a name of a method to the `CreateTask`. This approach is used in the demo application `1_HelloWorld`. The interesting point here is that you can declare this method in the same class from which the `CreateTask` is called. That way you can access all class fields and methods from the threaded code. Just keep in mind that you'll be doing this from another thread so make sure you protect shared access with locking!

```
procedure TfrmTestHelloWorld.RunHelloWorld(const task: IOmniTask);
begin
  //
end;

procedure TfrmTestHelloWorld.StartTask;
begin
  CreateTask(RunHelloWorld, 'HelloWorld').Run;
end;
```

In Delphi 2009 and newer you can also write the task code as an anonymous function.

```
CreateTask(
  procedure (const task: IOmniTask)
  begin
    //
  end,
  'HellowWorld').Run;
```

For all except the simplest tasks, you'll use the fourth approach as it will give you access to the true OmniThreadLibrary power (namely internal wait loop and message dispatching). To use it, you have to create a worker object deriving from the `TOmniWorker` class.

```
type
  THelloWorker = class(TOmniWorker)
  end;

procedure TfrmTestTwoWayHello.actStartHelloExecute(Sender: TObject);
begin
  FHelloTask :=
    CreateTask(THelloWorker.Create(), 'Hello').
    Run;
end;
```

4.3 IOmniTaskControl and IOmniTask interfaces

When you create a low-level task, OmniThreadLibrary returns a task controller interface `IOmniTaskControl`. This interface, which is defined in the OtlTaskControl unit, can be used to control the task from the owner's side. The task code, on the other hand, has access to another interface, `IOmniTask` (defined in the OtlTask unit), which can be used to communicate with the owner and manipulate the task itself. A picture in the Tasks vs. threads chapter shows the relationship between these interfaces.

This chapter deals mainly with these two interfaces. For the reference reasons, the `IOmniTaskControl` is reprinted here in full. In the rest of the chapter I'll just show relevant interface parts.

The `IOmniTask` interface is described at the end of this chapter.

```
type
  IOmniTaskControl = interface
```

```
    function  Alertable: IOmniTaskControl;
    function  CancelWith(const token: IOmniCancellationToken): IOmniTaskControl;
    function  ChainTo(const task: IOmniTaskControl;
      ignoreErrors: boolean = false): IOmniTaskControl;
    function  ClearTimer(timerID: integer): IOmniTaskControl;
    function  DetachException: Exception;
    function  Enforced(forceExecution: boolean = true): IOmniTaskControl;
    function  GetFatalException: Exception;
    function  GetParam: TOmniValueContainer;
    function  Invoke(const msgMethod: pointer): IOmniTaskControl; overload;
    function  Invoke(const msgMethod: pointer;
      msgData: array of const): IOmniTaskControl; overload;
    function  Invoke(const msgMethod: pointer;
      msgData: TOmniValue): IOmniTaskControl; overload;
    function  Invoke(const msgName: string): IOmniTaskControl; overload;
    function  Invoke(const msgName: string;
      msgData: array of const): IOmniTaskControl; overload;
    function  Invoke(const msgName: string;
      msgData: TOmniValue): IOmniTaskControl; overload;
    function  Invoke(remoteFunc: TOmniTaskControlInvokeFunction):
      IOmniTaskControl; overload;
    function  Invoke(remoteFunc: TOmniTaskControlInvokeFunctionEx):
      IOmniTaskControl; overload;
    function  Join(const group: IOmniTaskGroup): IOmniTaskControl;
    function  Leave(const group: IOmniTaskGroup): IOmniTaskControl;
    function  MonitorWith(const monitor: IOmniTaskControlMonitor):
      IOmniTaskControl;
    function  MsgWait(wakeMask: DWORD = QS_ALLEVENTS): IOmniTaskControl;
    function  NUMANode(numaNodeNumber: integer): IOmniTaskControl;
    function  OnMessage(eventDispatcher: TObject): IOmniTaskControl; overload;
    function  OnMessage(eventHandler: TOmniTaskMessageEvent): IOmniTaskControl; overload;
    function  OnMessage(msgID: word; eventHandler: TOmniTaskMessageEvent):
      IOmniTaskControl; overload;
    function  OnMessage(msgID: word; eventHandler: TOmniMessageExec):
      IOmniTaskControl; overload;
    function  OnMessage(eventHandler: TOmniOnMessageFunction):
      IOmniTaskControl; overload;
    function  OnMessage(msgID: word; eventHandler: TOmniOnMessageFunction):
      IOmniTaskControl; overload;
    function  OnTerminated(eventHandler: TOmniOnTerminatedFunction):
      IOmniTaskControl; overload;
    function  OnTerminated(eventHandler: TOmniOnTerminatedFunctionSimple):
      IOmniTaskControl; overload;
    function  OnTerminated(eventHandler: TOmniTaskTerminatedEvent):
      IOmniTaskControl; overload;
    function  ProcessorGroup(procGroupNumber: integer): IOmniTaskControl;
    function  RemoveMonitor: IOmniTaskControl;
    function  Run: IOmniTaskControl;
    function  Schedule(const threadPool: IOmniThreadPool = nil {default pool}):
      IOmniTaskControl;
    function  SetMonitor(hWindow: THandle): IOmniTaskControl;
    function  SetParameter(const paramName: string;
      const paramValue: TOmniValue): IOmniTaskControl; overload;
    function  SetParameter(const paramValue: TOmniValue):
      IOmniTaskControl; overload;
    function  SetParameters(const parameters: array of TOmniValue):
      IOmniTaskControl;
    function  SetPriority(threadPriority: TOTLThreadPriority): IOmniTaskControl;
    function  SetQueueSize(numMessages: integer): IOmniTaskControl;
    function  SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TOmniMessageID): IOmniTaskControl; overload;
    procedure SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TProc); overload;
    procedure SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TProc<integer>); overload;
    function  SetUserData(const idxData: TOmniValue;
      const value: TOmniValue): IOmniTaskControl;
    procedure Stop;
    function  Terminate(maxWait_ms: cardinal = INFINITE): boolean;
    function  TerminateWhen(event: THandle): IOmniTaskControl; overload;
```

```
73    function  TerminateWhen(token: IOmniCancellationToken):
74      IOmniTaskControl; overload;
75    function  Unobserved: IOmniTaskControl;
76    function  WaitFor(maxWait_ms: cardinal): boolean;
77    function  WaitForInit: boolean;
78    function  WithCounter(const counter: IOmniCounter): IOmniTaskControl;
79    function  WithLock(const lock: TSynchroObject;
80      autoDestroyLock: boolean = true): IOmniTaskControl; overload;
81    function  WithLock(const lock: IOmniCriticalSection):
82      IOmniTaskControl; overload;
83  //
84    property CancellationToken: IOmniCancellationToken
85      read GetCancellationToken;
86    property Comm: IOmniCommunicationEndpoint read GetComm;
87    property ExitCode: integer read GetExitCode;
88    property ExitMessage: string read GetExitMessage;
89    property FatalException: Exception read GetFatalException;
90    property Lock: TSynchroObject read GetLock;
91    property Name: string read GetName;
92    property Param: TOmniValueContainer read GetParam;
93    property UniqueID: int64 read GetUniqueID;
94    property UserData[const idxData: TOmniValue]: TOmniValue
95      read GetUserDataVal write SetUserDataVal;
96  end;
```

4.4 Task controller needs an owner

The `IOmniTaskController` interface returned from the `CreateTask` must always be stored in a variable/field with a scope that exceeds the lifetime of the background task. In other words, don't store a long-term background task interface in a local variable.

The simplest example of the wrong approach can be written in one line:

```
1 CreateTask(MyWorker).Run;
```

This code looks fine, but it doesn't work. In this case, the `IOmniTaskController` interface is stored in a hidden temporary variable which is destroyed at the end of the current method. This then causes the task controller to be destroyed which in turn causes the background task to be destroyed. Running this code would therefore just create and then destroy the task.

A common solution is to just store the interface in some field.

```
1 FTaskControl := CreateTask(MyWorker).Run;
```

When you don't need background worker anymore, you should terminate the task and free the task controller.

```
1 FTaskControl.Terminate;
2 FTaskControl := nil;
```

Another solution is to provide the task with an implicit owner. You can, for example, use the event monitor to monitor tasks lifetime or messages sent from the task and that will make the task owned by the monitor. The following code is therefore valid:

```
1 CreateTask(MyWorker).MonitorWith(eventMonitor).Run;
```

Yet another possibility is to call the `Unobserved` before the `Run`. This method makes the task being observed by an internal monitor.

```
1 CreateTask(MyWorker).Unobserved.Run;
```

When you use a thread pool to run a task, the thread pool acts as a task owner so there's

no need for an additional explicit owner.

```
procedure Beep(const task: IOmniTask);
begin
  MessageBeep(MB_ICONEXCLAMATION);
end;

CreateTask(Beep, 'Beep').Schedule;
```

4.5 Communication subsystem

As it is explained in the Locking vs. messaging section, OmniThreadLibrary automatically creates a communication channel between the task controller and the task and exposes it through the `Comm` property. The communication channel is not exclusive to the OmniThreadLibrary; you could use it equally well from a `TThread`-based multithreading code.

```
property Comm: IOmniCommunicationEndpoint read GetComm;
```

The `IOmniCommunicationEndpoint` interface exposes a simple interface for sending and receiving messages.

```
type
  TOmniMessage = record
    MsgID : word;
    MsgData: TOmniValue;
    constructor Create(aMsgID: word; aMsgData: TOmniValue); overload;
    constructor Create(aMsgID: word); overload;
  end;

  IOmniCommunicationEndpoint = interface
    function  Receive(var msg: TOmniMessage): boolean; overload;
    function  Receive(var msgID: word; var msgData: TOmniValue): boolean; overload;
    function  ReceiveWait(var msg: TOmniMessage; timeout_ms: cardinal): boolean; overload;
    function  ReceiveWait(var msgID: word; var msgData: TOmniValue;
      timeout_ms: cardinal): boolean; overload;
    procedure Send(const msg: TOmniMessage); overload;
    procedure Send(msgID: word); overload;
    procedure Send(msgID: word; msgData: array of const); overload;
    procedure Send(msgID: word; msgData: TOmniValue); overload;
    function  SendWait(msgID: word;
      timeout_ms: cardinal = CMaxSendWaitTime_ms): boolean; overload;
    function  SendWait(msgID: word; msgData: TOmniValue;
      timeout_ms: cardinal = CMaxSendWaitTime_ms): boolean; overload;
    property NewMessageEvent: THandle read GetNewMessageEvent;
    property OtherEndpoint: IOmniCommunicationEndpoint read GetOtherEndpoint;
    property Reader: TOmniMessageQueue read GetReader;
    property Writer: TOmniMessageQueue read GetWriter;
  end;
```

- `Receive`

  Both variants of `Receive` return the first message from the message queue, either as a `TOmniMessage` record or as a (message ID, message data) pair. Data is always passed as a `TOmniValue` record.

  The function returns `True` if a message was returned, `False` if the message queue is empty.

- `ReceiveWait`

  These two variations of the `Receive` allow you to specify the maximum timeout (in milliseconds) you are willing to wait for the next message. Timeout of 0 milliseconds makes the function behave just like the `Receive`. Special timeout value `INFINITE` (defined in the Windows unit) will make the function wait until a message is available.

The function returns `True` if a message was returned, `False` if the message queue is still empty after the timeout.

- `Send`

  Four overloaded versions of `Send` all write a message to the message queue and raise an exception if the queue is full. [Message queue size defaults to 1000 elements and can be increased by calling the `OmniTaskControl.SetQueueSize` before the communication channel is used for the first time.]

  The `Send(msgID: word)` version sends an empty message data (`TOmniValue.Null`).

  The `Send(msgID: word; msgData: array of const)` version packs the data array into one `TOmniValue` value by calling `TOmniValue.Create(msgData)`.

- `SendWait`

  These two variations of the `Send` method allow you to specify the maximum timeout (in milliseconds) you are willing to wait if a message queue is full and there's no place for the messages. The timeout of 0 ms makes the function behave just like the `Send`. A timeout of `INFINITE` milliseconds is also supported.

  The function returns `True` if message was successfully sent, `False` if the message queue is still full after the timeout.

- `NewMessageEvent`

  This property returns Windows event which is signalled every time new data is inserted in the queue. This event is not created until the code accesses the `NewMessageEvent` property for the first time.

- `OtherEndpoint`

  Returns the other end of the communication channel (task's end if accessed through the `IOmniTaskControl.Comm` and task controller's end if accessed through the `IOmniTask.Comm` interface).

- `Reader`

  Returns the input queue associated with this endpoint.

- `Writer`

  Returns the output queue associated with this endpoint.

In versions up to [3.04a], both `SendWait` and `ReceiveWait` were designed to be used from only one thread at a time. Since OmniThreadLibrary [3.04b] they are both fully thread-safe and can be used from multiple producers and consumers at the same time.

For practical examples on communication channel usage, see the Communication subsection of simple tasks and `TOmniWorker` tasks sections.

Communication message queue is implemented using the Bounded Queue structure.

4.6 Processor groups and NUMA nodes

On a system with multiple processor groups you can use `ProcessorGroup` [3.06] function to specify a processor group this task should run on.

On a system with multiple NUMA nodes you can use NUMANode [3.06] function to specify a NUMA node this task should run on.

When a task is not started directly (Run) but executed via thread pool (Schedule), IOmniThreadPool.ProcessorGroups and IOmniThreadPool.NUMANodes should be used instead.

An information about existing processor groups and NUMA nodes can be accessed through the Environment object.

Demo 64_ProcessorGroups_NUMA demonstrates the use of ProcessorGroup and NUMANode functions.

4.7 Thread pooling

Starting a new thread in the Windows OS is not very fast operation. If you are frequently scheduling background tasks, the overhead of creating new threads can significantly impact your program. To solve this, OmniThreadLibrary implements a thread pool, which is a basically a cache for threads.

All high-level tasks are executed in a thread pool unless NoThreadPool is used.

You don't run a task in a thread pool but schedule it by calling the Schedule method. A very short example of a scheduled task would be:

```
1 procedure Beep(const task: IOmniTask);
2 begin
3   MessageBeep(MB_ICONEXCLAMATION);
4 end;
5
6 CreateTask(Beep, 'Beep').Schedule;
```

A thread pool is created by calling the CreateThreadPool function. A thread pool should have a name (you can set it to an empty string) which is used as part of thread pool management thread's name.

You can also use the default GlobalOmniThreadPool pool which is created on the first use.

```
1 function CreateThreadPool(const threadPoolName: string): IOmniThreadPool;
2
3 function GlobalOmniThreadPool: IOmniThreadPool;
```

High-level tasks use their own pool called GlobalParallelPool.

All thread pool-related code is stored in the OtlThreadPool unit.

See also demo 11_ThreadPool.

4.7.1 Execution flow

When you schedule a task into a thread pool, it merely enters a queue. The thread pool management thread detects this and tries to start this task in an already existing but idle thread. If there is no such thread, it tries to create a new thread (you can limit the maximum number of concurrent threads so this may not always succeed) and run the task in it.

When a task finishes execution, the thread it was running in is put into an idle state and may be reused for execution of new tasks.

Next section explains the various configuration options implemented by the thread pool.

4.7.2 IOmniThreadPool interface

```
type
  TOTPThreadDataFactoryFunction = function: IInterface;
  TOTPThreadDataFactoryMethod = function: IInterface of object;

  IOmniThreadPool = interface
    function  Cancel(taskID: int64): boolean; overload;
    function  Cancel(taskID: int64; signalCancellationToken: boolean): boolean; overload;
    procedure CancelAll; overload;
    procedure CancelAll(signalCancellationToken: boolean); overload;
    function  CountExecuting: integer;
    function  CountQueued: integer;
    function  IsIdle: boolean;
    function  MonitorWith(const monitor: IOmniThreadPoolMonitor): IOmniThreadPool;
    function  RemoveMonitor: IOmniThreadPool;
    function  SetMonitor(hWindow: THandle): IOmniThreadPool;
    procedure SetThreadDataFactory(const value: TOTPThreadDataFactoryMethod); overload;
    procedure SetThreadDataFactory(const value: TOTPThreadDataFactoryFunction); overload;
    property Asy_OnUnhandledWorkerException: TOTPUnhandledWorkerException read
      GetAsy_OnUnhandledWorkerException write SetAsy_OnUnhandledWorkerException;
    property Affinity: IOmniIntegerSet read GetAffinity;
    property IdleWorkerThreadTimeout_sec: integer read GetIdleWorkerThreadTimeout_sec
      write SetIdleWorkerThreadTimeout_sec;
    property MaxExecuting: integer read GetMaxExecuting write SetMaxExecuting;
    property MaxQueued: integer read GetMaxQueued write SetMaxQueued;
    property MaxQueuedTime_sec: integer read GetMaxQueuedTime_sec write
      SetMaxQueuedTime_sec;
    property MinWorkers: integer read GetMinWorkers write SetMinWorkers;
    property Name: string read GetName write SetName;
    property NumCores: integer read GetNumCores;
    property Options: TOmniThreadPoolOptions read GetOptions write SetOptions;
    property UniqueID: int64 read GetUniqueID;
    property WaitOnTerminate_sec: integer read GetWaitOnTerminate_sec
      write SetWaitOnTerminate_sec;
    {$IFDEF OTL_NUMASupport}
    property ProcessorGroups: IOmniIntegerSet read GetProcessorGroups write
      SetProcessorGroups;
    property NUMANodes: IOmniIntegerSet read GetNUMANodes write SetNUMANodes;
    {$ENDIF OTL_NUMASupport}
  end;
```

Methods

- `Cancel`

  Cancels a task with specified unique ID by calling task's `Terminate` method. If a task does not stop within `WaitOnTerminate_sec` seconds, the thread that is running the task will be killed by calling the `TerminateThread` Windows function.

  Since [3.07.2], `Cancel` signals task's cancellation token before calling Terminate. Old behaviour (without signalling cancellation token) can be achieved by calling overloaded version accepting the `signalCancellationToken` parameter or by setting `Option` to `[tpoPreventCancellationTokenOnCancel]`.

  If a task is not yet executing, it will simply be removed from the input queue.

- `CancelAll`

  Cancels all tasks, waiting and running.

- `CountExecuting`

  Returns number of currently executing tasks.

- `CountQueued`

  Returns number of queued (waiting) tasks.

- `IsIdle`

  Returns `True` when thread pool has no work to do.

- `MonitorWith`

  Attaches external monitor to the thread pool.

- `RemoveMonitor`

  Detaches external monitor from the thread pool.

- `SetMonitor`

  This is internal function, called from the external monitor when attaching to the thread pool.

- `SetThreadDataFactory`

  Associates a data factory function with the thread pool. A data factory can be used to create thread local data whenever a new thread is created.

  [3.07] If `ThreadDataFactory.Execute` throws an exception, that exception is caught, ignored and `ThreadData` is set to nil.

  The Building a connection pool example contains more information on this topic.

Properties

- `Affinity` [3.06]

  Provides a set of all processors that are used for executing tasks in this thread pool. By changing this set you can specify which processors will be used to run tasks scheduled to this thread pool.

- `IdleWorkerThreadTimeout_sec`

  Specifies the maximum time a thread can spend in an idle state. After that, the thread will be terminated. By default, this value is set to 10 seconds.

  By setting this value to 0, idle threads are never terminated.

  OmniThreadLibrary will always keep `MinWorkers` threads alive, even if they are idle for more than `IdleWorkerThreadTimeout_sec` seconds.

- `MaxExecuting`

  Specifies the maximum number of working threads in the thread pool. Initially, this value is set to the number of cores current process is allowed to run on.

  By setting this value to 0 you can prevent the thread pool from creating any threads. You can use this to temporarily stop a thread pool without destroying it.

  If `MaxExecuting` is set to -1, number of running threads is only limited by the implementation.

Starting with [3.04], a maximum number of concurrent threads in a thread pool is not limited. Before that, it was limited to 60 concurrent threads.

- `MaxQueued`

  Specifies maximum number of tasks waiting in the input queue. If there are already `MaxQueued` tasks in the queue and a new task is scheduled, it will be immediately rejected (see Monitoring thread pool operations for more information).

  If `MaxQueued` is set to 0 (which is the default value), the size of the input queue is not limited.

- `MaxQueuedTime_sec`

  Specifies maximum time a task will wait in the input queue before it is rejected (see Monitoring thread pool operations for more information).

  If set to 0 (which is the default value), the time a task can spend in the input queue is not limited.

- `MinWorkers`

  Specifies the minimum number of threads that should be created in any moment. Both idle and working threads are counted. By default this value is set to 0.

  Since [3.05] setting this value to a positive number will create specified number of worker threads even if no tasks are waiting for the execution.

- `Name`

  Specifies the name of the thread pool.

- `NumCores`

  Returns number of cores this pool uses for running tasks. Changing `Affinity`, `ProcessorGroups`, or `NUMANodes` properties may modify this value.

- `Options`

  Contains set of options that govern how thread pool operates internally. See `Cancel` for more information.

- `UniqueID`

  Gives readonly access to the unique ID associated with the pool. This value is guaranteed to be greater than 0.

- `WaitOnTerminate_sec`

  When a pooled task is terminated after a `Cancel` or `CancelAll` is called, a thread pool manager will wait up to this number of seconds for the task to stop execution. After that, it will kill the thread with the `TerminateThread` call.

  By default this value is set to 30 seconds.

- `Asy_OnUnhandledWorkerException` [3.06]

  Before version 3.06, unhandled exceptions in the code handling the task execution were lost. Now, they are passed up to the `IOmniThreadPool`. If the property `Asy_OnUnhandledWorkerException` is set, such exception will be passed to the event handler

and application should react to it. The only safe way at that point is to log the error and terminate the application.

4.7.3 Task exit code

Various reasons for task termination are signalled through its ExitCode property. Following thread pool-specific exit codes are defined.

- EXIT_THREADPOOL_QUEUE_TOO_LONG

  Task was rejected because the input queue already contains at least MaxQueued tasks.

- EXIT_THREADPOOL_STALE_TASK

  Task was rejected because it was waiting in the input queue for more than MaxQueuedTime_sec seconds.

- EXIT_THREADPOOL_CANCELLED

  Task was cancelled with a Cancel or CancelAll call.

- EXIT_THREADPOOL_INTERNAL_ERROR

  This exit code is currently not in use.

4.7.4 Monitoring thread pool operations

An event monitor component can be used to monitor thread pool events. First you have to attach it to the thread pool by calling the MonitorWith method. After that, following event monitor event handlers can be used:

```
type
  TOmniMonitorPoolThreadEvent = procedure(const pool: IOmniThreadPool;
    threadID: integer) of object;
  TOmniMonitorPoolWorkItemEvent = procedure(const pool: IOmniThreadPool;
    taskID: int64) of object;

  TOmniEventMonitor = class(TComponent,
                            IOmniTaskControlMonitor,
                            IOmniThreadPoolMonitor)
  public
    property OnPoolThreadCreated: TOmniMonitorPoolThreadEvent;
    property OnPoolThreadDestroying: TOmniMonitorPoolThreadEvent;
    property OnPoolThreadKilled: TOmniMonitorPoolThreadEvent;
    property OnPoolWorkItemCompleted: TOmniMonitorPoolWorkItemEvent;
  end;
```

- OnPoolThreadCreated

  This event is called whenever a new thread is created in the pool.

- OnPoolThreadDestroying

  This event is called just before a thread is destroyed.

- OnPollThreadKilled

  This event is called when a thread is killed because the task did not stop in the allowed time.

- OnPoolWorkItemCompleted

  This event is called when a task is removed from the thread pool. This can occur on task completion, when a task is removed before it has even started (Cancel, CancelAll)

and when a thread containing a task is killed because the `Cancel` was called and the task didn't stop in the allowed time.

Single event monitor can be used to monitor multiple thread pools.

4.7.5 Processor groups and NUMA nodes

Through the property `ProcessorGroups` you can specify all processor groups the tasks can be scheduled to. By default this set is empty which means that tasks will only be scheduled to the default processor group.

By setting the property `NUMANodes` you can specify all NUMA nodes the tasks can be scheduled to. By default this set is empty which means that tasks will only be scheduled to the default NUMA node.

An information about existing processor groups and NUMA nodes can be accessed through the `Environment` object.

Demo `64_ProcessorGroups_NUMA` demonstrates the use of `ProcessorGroup` and `NUMANode` functions.

4.8 Lock-free collections

OmniThreadLibrary implements three lock-free data structures suitable for low-level usage – bounded stack, bounded queue and dynamic queue. Bounded queue is used inside the OmniThreadLibrary for messaging and dynamic queue is used as a basis of the blocking collection.

All three data structures are fully thread-safe. They support multiple simultaneous readers and writers. They are implemented in the OtlContainers unit.

Another lock-free data structure, a message queue, is defined in the OtlComm unit and is mostly intended for internal operation (such as sending messages to and from thread) although it can also be used for other tasks. An example of such usage is shown in the Using message queue with a `TThread` worker chapter.

> The term lock-free is not well defined (and not even universally accepted). In the context of this book lock-free means that the synchronisation between threads is not achieved with the user- or kernel-level synchronisation primitives such as critical sections, but with bus-locking CPU instructions. With modern CPU architectures this approach is much faster than locking on the operating system level.

See also demos `10_Containers` and `32_Queue`.

4.8.1 Bounded Stack

The bounded stack structure is a very fast stack with limited length. The core of the implementation is stored in the `TOmniBaseBoundedStack` class.

Derived class `TOmniBoundedStack` adds support for external observers. Both classes implement the same interface – `IOmniStack` – so you can code against the class or against the interface.

```
1 type
2   IOmniStack = interface
```

```
3     procedure Empty;
4     procedure Initialize(numElements, elementSize: integer);
5     function  IsEmpty: boolean;
6     function  IsFull: boolean;
7     function  Pop(var value): boolean;
8     function  Push(const value): boolean;
9   end;
10
11  TOmniBaseBoundedStack = class(TInterfacedObject, IOmniStack)
12  public
13    destructor Destroy; override;
14    procedure Empty;
15    procedure Initialize(numElements, elementSize: integer); virtual;
16    function  IsEmpty: boolean; inline;
17    function  IsFull: boolean; inline;
18    function  Pop(var value): boolean;
19    function  Push(const value): boolean;
20    property  ElementSize: integer read obsElementSize;
21    property  NumElements: integer read obsNumElements;
22  end;
23
24  TOmniBoundedStack = class(TOmniBaseBoundedStack)
25  public
26    constructor Create(numElements, elementSize: integer;
27      partlyEmptyLoadFactor: real = CPartlyEmptyLoadFactor;
28      almostFullLoadFactor: real = CAlmostFullLoadFactor);
29    destructor  Destroy; override;
30    function Pop(var value): boolean;
31    function Push(const value): boolean;
32    property ContainerSubject: TOmniContainerSubject read osContainerSubject;
33  end;
```

- `Empty`

  Empties the stack.

- `Initialize`

  Initializes the stack for maximum `numElements` elements of size `elementSize`.

- `IsEmpty`

  Returns `True` when the stack is empty.

- `IsFull`

  Returns `True` when the stack is full.

- `Pop`

  Takes one value from the stack and returns `True` if the stack was not empty before the operation.

- `Push`

  Puts one value on the stack and returns `True` if there was place for the value (the stack was not full before the operation).

- `ElementSize`

  Returns the size of the stack element as set in the `Initialize` call.

- `NumElements`

  Returns maximum number of elements in the stack as set in the `Initialize` call.

- `ContainerSubject`

  Provides a point for attaching external observers as described in the Observing lock-free collections section.

4.8.2 Bounded queue

The bounded queue structure is a very fast queue with limited length.

The core of the implementation is stored in the `TOmniBaseBoundedQueue` class. Derived class `TOmniBoundedQueue` adds support for external observers. Both classes implement the same interface – `IOmniQueue` – so you can code against the class or against the interface.

```
type
  IOmniQueue = interface
    function  Dequeue(var value): boolean;
    procedure Empty;
    function  Enqueue(const value): boolean;
    procedure Initialize(numElements, elementSize: integer);
    function  IsEmpty: boolean;
    function  IsFull: boolean;
  end;

  TOmniBaseBoundedQueue = class(TInterfacedObject, IOmniQueue)
  public
    destructor Destroy; override;
    function  Dequeue(var value): boolean;
    procedure Empty;
    function  Enqueue(const value): boolean;
    procedure Initialize(numElements, elementSize: integer); virtual;
    function  IsEmpty: boolean;
    function  IsFull: boolean;
    property  ElementSize: integer read obqElementSize;
    property  NumElements: integer read obqNumElements;
  end;

  TOmniBoundedQueue = class(TOmniBaseBoundedQueue)
  public
    constructor Create(numElements, elementSize: integer;
      partlyEmptyLoadFactor: real = CPartlyEmptyLoadFactor;
      almostFullLoadFactor: real = CAlmostFullLoadFactor);
    destructor  Destroy; override;
    function  Dequeue(var value): boolean;
    function  Enqueue(const value): boolean;
    property  ContainerSubject: TOmniContainerSubject read oqContainerSubject;
  end;
```

- `Empty`

  Empties the stack.

- `Dequeue`

  Takes one value from the queue’ s head and returns `True` if the queue was not empty before the operation.

- `Enqueue`

  Inserts one value on the queue’ s tail and returns `True` if there was place for the value (the queue was not full before the operation).

- `Initialize`

  Initializes the queue for maximum `numElements` elements of size `elementSize`.

- `IsEmpty`

  Returns `True` when the queue is empty.

- `IsFull`

  Returns `True` when the queue is full.

- `ElementSize`

  Returns size of the queue element as set in the `Initialize` call.

- `NumElements`

  Returns maximum number of elements in the queue as set in the `Initialize` call.

- `ContainerSubject`

  Provides a point for attaching external observers as described in the Observing lock-free collections section.

4.8.3 Message queue

The `TOmniMessageQueue` is just a thin wrapper around the bounded queue data structure. An element of this queue is a (message ID, message data) pair, stored in a `TOmniMessage` record.

This class greatly simplifies creating and attaching event and window observers.

```
type
  TOmniMessage = record
    MsgID : word;
    MsgData: TOmniValue;
    constructor Create(aMsgID: word; aMsgData: TOmniValue); overload;
    constructor Create(aMsgID: word); overload;
  end;

  TOmniContainerWindowsEventObserver = class(TOmniContainerObserver)
  public
    function  GetEvent: THandle; virtual; abstract;
  end;

  TOmniMessageQueueMessageEvent =
    procedure(Sender: TObject; const msg: TOmniMessage) of object;

  TOmniMessageQueue = class(TOmniBoundedQueue)
  public
    constructor Create(numMessages: integer;
      createEventObserver: boolean = true); reintroduce;
    destructor  Destroy; override;
    function  Dequeue: TOmniMessage; reintroduce;
    function  Enqueue(const value: TOmniMessage): boolean; reintroduce;
    procedure Empty;
    function  GetNewMessageEvent: THandle;
    function  TryDequeue(var msg: TOmniMessage): boolean; reintroduce;
    property EventObserver: TOmniContainerWindowsEventObserver
      read mqWinEventObserver;
    property OnMessage: TOmniMessageQueueMessageEvent
      read mqWinMsgObserver.OnMessage write SetOnMessage;
  end;
```

`TOmniMessageQueue.Create` creates an event observer unless the second parameter (`createEventObserver`) is set to `False`. It is created with the `coiNotifyOnAllInserts` interest meaning that an event (accessible through the `GetNewMessageEvent` function) is signalled each time an

element (a message) is added to the queue. The observer itself is accessible through the `EventObserver` property.

You can also easily create a window message observer by attaching an event handler to the `OnMessage` property. This observer is also created with the `coiNotifyOnAllInserts` interest which causes the `OnMessage` event handler to be called each time an element (a message) is added to the queue. You can destroy this observer at any time by assigning a `nil` value to the `OnMessage` event.

For an example, see chapter Using message queue with a `TThread` worker.

4.8.4 Dynamic queue

The dynamic queue is a fast queue with unlimited length. It can grow as required as the data used to store elements is dynamically allocated.

The core of the implementation is stored in the `TOmniBaseQueue` class. Derived class `TOmniQueue` adds support for external observers. Both structures store `TOmniValue` elements.

```
type
  TOmniBaseQueue = class
  ...
  public
    constructor Create(blockSize: integer = 65536; numCachedBlocks: integer = 4);
    destructor  Destroy; override;
    function  Dequeue: TOmniValue;
    procedure Enqueue(const value: TOmniValue);
    function  IsEmpty: boolean;
    function  TryDequeue(var value: TOmniValue): boolean;
  end;

  TOmniQueue = class(TOmniBaseQueue)
  ...
  public
    function  Dequeue: TOmniValue;
    procedure Enqueue(const value: TOmniValue);
    function  TryDequeue(var value: TOmniValue): boolean;
    property ContainerSubject: TOmniContainerSubject read ocContainerSubject;
  end;
```

- `Create`

  Creates a queue object with a specified page size (`blockSize`) where `numCachedBlocks` are always preserved for future use. Defaults (`65536` and `4`) should be appropriate for most scenarios.

- `Dequeue`

  Takes one element from queue’s head and returns it. If the queue is empty, an exception is raised.

- `Enqueue`

  Inserts an element on the queue’s tail.

- `IsEmpty`

  Returns `True` when the queue is empty.

- `TryDequeue`

Takes one element from queue's head and returns it in the `value` parameter. Returns `True` if an element was returned (the queue was not empty before the operation).

- `ContainerSubject`

  Provides a point for attaching external observers as described in the Observing lock-free collections section.

4.8.5 Observing lock-free collections

OmniThreadLibrary data structures support the observer design pattern. Each structure can be observed by multiple observers at the same time. Supporting code and two observer implementations are stored in the OtlContainerObserver unit.

Current architecture supports four different kinds of events that can be observed:

```
type
  ///<summary>All possible actions observer can take interest in.</summary>
  TOmniContainerObserverInterest = (
    //Interests with permanent subscription:
    coiNotifyOnAllInserts, coiNotifyOnAllRemoves,
    //Interests with one-shot subscription:
    coiNotifyOnPartlyEmpty, coiNotifyOnAlmostFull
  );
```

- `coiNotifyOnAllInserts`

  Observer is notified whenever a data element is inserted into the structure.

- `coiNotifyOnAllRemoves`

  Observer is notified whenever a data element is removed from the structure.

- `coiNotifyOnPartlyEmpty`

  Observer is notified whenever a data usage drops below the `partlyEmptyLoadFactor` (parameter of the data structure constructor, 80% by default). This event is only supported for bounded structures.

  This event can only be observed once. After that you should destroy the observer and (if required) create another one and attach it to the data structure.

- `coiNotifyOnAlmostFull`

  Observer is notified whenever a data usage rises above the `almostFullLoadFactor` (parameter of the data structure constructor, 90% by default). This event is only supported for bounded structures.

  This event can only be observed once. After that you should destroy the observer and (if required) create another one and attach it to the data structure.

The OtlContainerObserver unit implements event and message observers.

```
  TOmniContainerWindowsEventObserver = class(TOmniContainerObserver)
  public
    function  GetEvent: THandle; virtual; abstract;
  end;

  TOmniContainerWindowsMessageObserver = class(TOmniContainerObserver)
  strict protected
```

```
    function  GetHandle: THandle; virtual; abstract;
  public
    procedure Send(aMessage: cardinal; wParam, lParam: integer);
      virtual; abstract;
    property Handle: THandle read GetHandle;
  end;

  function CreateContainerWindowsEventObserver(externalEvent: THandle = 0):
    TOmniContainerWindowsEventObserver;

  function CreateContainerWindowsMessageObserver(hWindow: THandle;
    msg: cardinal; wParam, lParam: integer):
    TOmniContainerWindowsMessageObserver;
```

The event observer `TOmniContainerWindowsEventObserver` raises an event every time the observed event occurs.

The message observer `TOmniContainerWindowsMessageObserver` sends a message to a window every time the observed event occurs.

4.8.5.1 Examples

Create and attach the event observer:

```
FObserver := CreateContainerWindowsEventObserver;
FCollection.ContainerSubject.Attach(FObserver, coiNotifyOnAllInserts);
```

Access the observer event so you can wait on it:

```
FEvent := FObserver.GetEvent;
```

Detach and destroy the observer:

```
FCollection.ContainerSubject.Detach(FObserver, coiNotifyOnAllInserts);
FreeAndNil(FObserver);
```

Create and attach the message observer:

```
FWindow := DSiAllocateHWnd(ObserverWndProc);
FObserver := CreateContainerWindowsMessageObserver(
  FWindow, MSG_ITEM_INSERTED, 0, 0);
FWorker.Output.ContainerSubject.Attach(FObserver, coiNotifyOnAllInserts);
```

Process observer messages:

```
procedure ObserverWndProc(var message: TMessage);
var
  ovWorkItem: TOmniValue;
  workItem : IOmniWorkItem;
begin
  if message.Msg = MSG_ITEM_INSERTED then begin
    //...
    message.Result := Ord(true);
  end
  else
    message.Result := DefWindowProc(FWindow, message.Msg,
      message.WParam, message.LParam);
end;
```

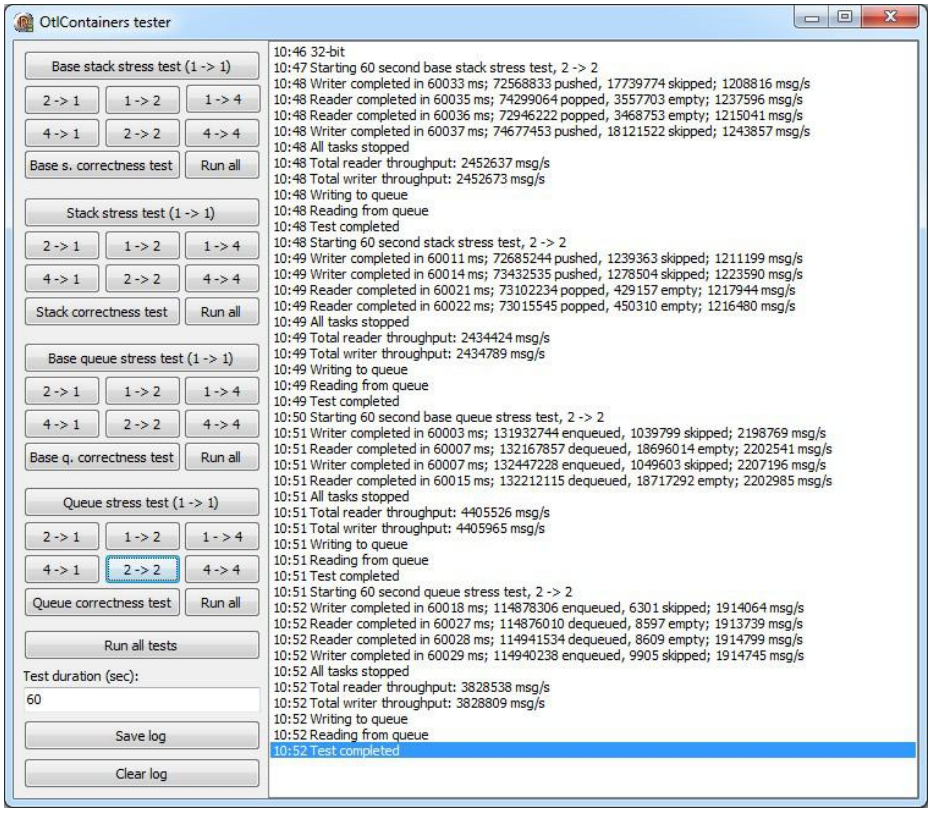
Detach and destroy the observer:

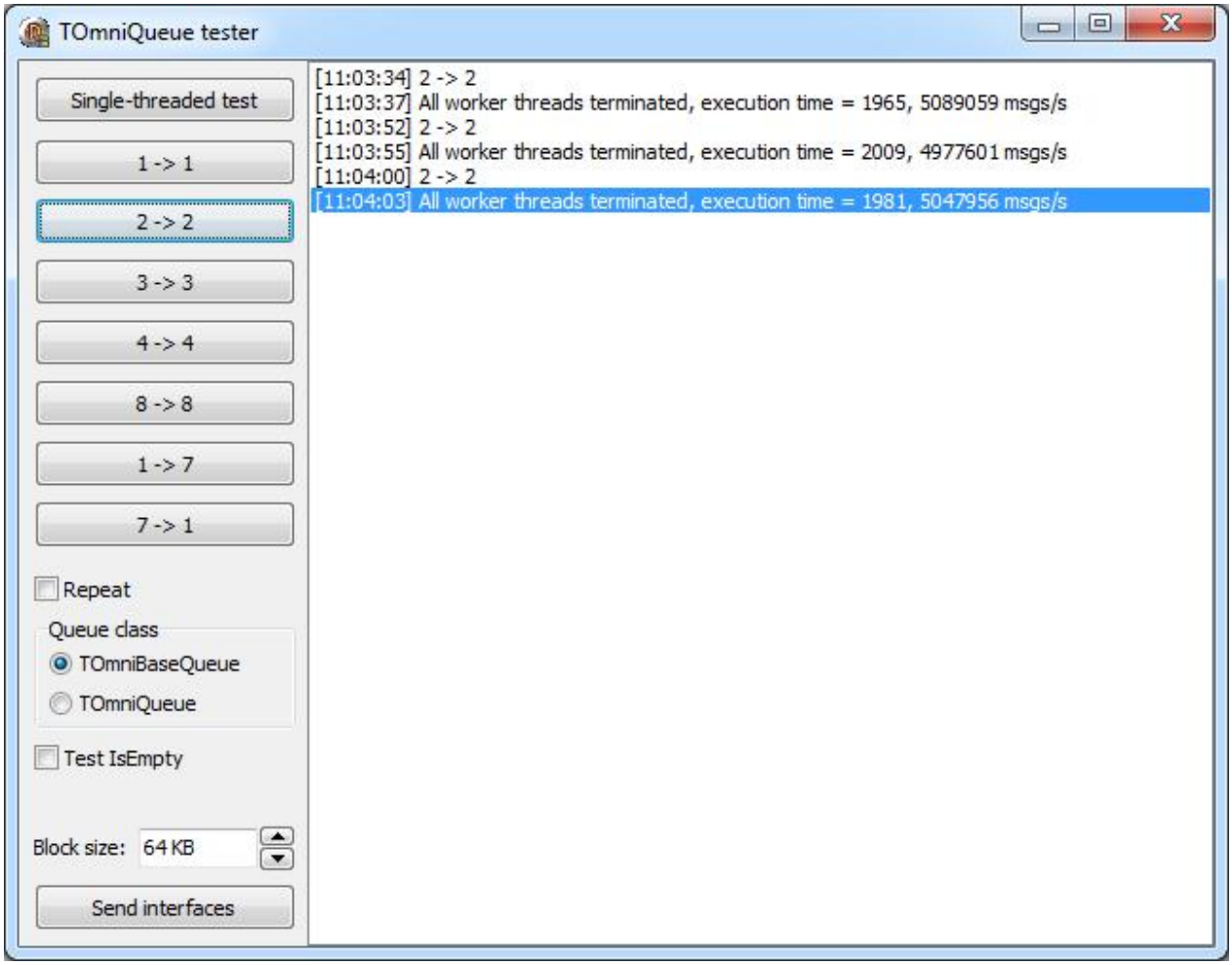
```
FWorker.Output.ContainerSubject.Detach(FObserver, coiNotifyOnAllInserts);
FreeAndNil(FObserver);
DSiDeallocateHWnd(FWindow);
```

4.8.6 Benchmarks

OmniThreadLibrary contains two demos that can be used to measure the performance of the lock-free structures. Bounded structures are benchmarked in the `10_Containers` demo and dynamic queue is benchmarked in the `32_Queue` demo.

Following results were measured on 4-core i7-2630QM running at 2 GHz. As you can see, lock-free structures can transfer from 2,5 to 5 million messages per second.





4.9 Event monitor

While the OmniThreadLibrary is mostly a code-oriented framework, it also contains one package (stored in the packages subfolder) with one component (`TOmniEventMonitor`). This component supports Win32 and Win64 projects and contains some events that can be used to monitor task and thread lifecycle.

```
type
  TOmniMonitorTaskEvent = procedure(const task: IOmniTaskControl) of object;
  TOmniMonitorTaskMessageEvent = procedure(const task: IOmniTaskControl;
    const msg: TOmniMessage) of object;
  TOmniMonitorPoolThreadEvent = procedure(const pool: IOmniThreadPool;
    threadID: integer) of object;
  TOmniMonitorPoolWorkItemEvent = procedure(const pool: IOmniThreadPool;
    taskID: int64) of object;

  TOmniEventMonitor = class(TComponent,
                            IOmniTaskControlMonitor,
                            IOmniThreadPoolMonitor)
  published
    property OnPoolThreadCreated: TOmniMonitorPoolThreadEvent;
    property OnPoolThreadDestroying: TOmniMonitorPoolThreadEvent;
    property OnPoolThreadKilled: TOmniMonitorPoolThreadEvent;
    property OnPoolWorkItemCompleted: TOmniMonitorPoolWorkItemEvent;
    property OnTaskMessage: TOmniMonitorTaskMessageEvent;
    property OnTaskTerminated: TOmniMonitorTaskEvent;
    property OnTaskUndeliveredMessage: TOmniMonitorTaskMessageEvent;
  end;
```

The first four events (`OnPoolThread*`) are used to monitor thread pool events. They are described in the Monitoring thread pool operations section.

The other three events are used to monitor tasks attached to this monitor. See the MonitorWith section for more information.

Event monitor can be dropped on a form or created in the code (see demo 2_TwoWayHello for an example).

4.10 Simple tasks

This part of the book describes properties and methods of the IOmniTaskControl interface that are useful with all four kinds of tasks. Next section, TOmniWorker tasks covers only parts that are useful for TOmniWorker-based tasks.

4.10.1 Name

```
1 property Name: string read GetName;
```

The Name property returns the task name as set in the CreateTask call.

4.10.2 UniqueID

```
1 property UniqueID: int64 read GetUniqueID;
```

The UniqueID property returns task's unique ID. This ID is generated automatically when a task is created and is guaranteed to be unique and greater than zero.

4.10.3 Parameters

```
1 function SetParameter(const paramName: string;
2   const paramValue: TOmniValue): IOmniTaskControl; overload;
3 function SetParameter(const paramValue: TOmniValue): IOmniTaskControl; overload;
4 function SetParameters(const parameters: array of TOmniValue): IOmniTaskControl;
5 property Param: TOmniValueContainer read GetParam;
```

Task controller can set parameters for the task by calling the SetParameter and SetParameters methods. Parameters must be set before the task is started or scheduled (IOW, before the Run or Schedule are called).

Parameters can have names or they can be accessed by an index number.

In the former approach you have to pass in a name and a value for each parameter. You can set multiple parameters in one call by calling SetParameters and providing it with an array of (name, value) pairs.

```
1 taskControl := CreateTask(MyTask);
2 taskControl.SetParameter('Initial value', '42');
3 taskControl.SetParameters(['From', 0, 'To', 99]);
```

Both the controller and the task can access parameters through the Param property.

```
1 task.Param['Initial value'] // '42'
2 task.Param['From'] // 0
3 task.Param['To'] // 99
```

Another approach is to just set parameters one by one by calling SetParameter repeatedly.

```
1 taskControl := CreateTask(MyTask);
2 taskControl.SetParameter('42');
3 taskControl.SetParameter(0);
4 taskControl.SetParameter(99);
```

In this case, task can index the `Param` property with a 0-based index to access parameters.

```
1 task.Param[0] // '42'
2 task.Param[1] // 0
3 task.Param[2] // 99
```

Parameter storage is implemented in the `TOmniValueContainer` class.

4.10.4 Termination

A simple background task can function either as a single-shot or as a long term operation. In the former case there's no need for stopping the task as it will stop by itself but in the latter case we would need to tell the task to stop at some point.

To stop a task, call its `Terminate` method. It will wait up to `maxWait_ms` milliseconds for the task to exit, after which it will kill the thread running the task.

```
1 procedure Stop;
2 function Terminate(maxWait_ms: cardinal = INFINITE): boolean;
3 function WaitFor(maxWait_ms: cardinal): boolean;
```

Simple tasks should occasionally check either the `Terminated` function (it will return `True` once the `Terminate` has been called) or the `TerminateEvent` event (it will become signalled once the `Terminate` has been called).

The two examples below demonstrate how to write a task that does nothing except wait to be terminated.

```
1 procedure TerminateTask1(const task: IOmniTask);
2 begin
3   while WaitForSingleObject(task.TerminateEvent, 1000) = WAIT_TIMEOUT do begin
4     // some periodic task
5   end;
6 end;
7
8 procedure TerminateTask2(const task: IOmniTask);
9 begin
10  while not task.Terminated do begin
11    // some periodic task
12    Sleep(1000);
13  end;
14 end;
```

If you just want to tell the task to stop (without waiting for its termination), you can call the `Stop` method. You can then check the tasks status by calling the `WaitFor` function which will wait at most `maxWait_ms` milliseconds for the task to stop and then return `True` if the task has stopped.

`WaitFor` is also useful if your task stops automatically or if you implement some other mechanism to stop the task.

Sometimes you may want to be notified when the task has terminated. One way to do that is to attach the task to a monitor and monitor the `OnTaskTerminated` event. Another is to write a termination event handler.

```
1 type
2   TOmniOnTerminatedFunction = reference to procedure(const task: IOmniTaskControl);
3   TOmniOnTerminatedFunctionSimple = reference to procedure;
4   TOmniTaskTerminatedEvent = procedure(const task: IOmniTaskControl) of object;
5
6 function OnTerminated(eventHandler: TOmniOnTerminatedFunction):
7   IOmniTaskControl; overload;
```

```
function OnTerminated(eventHandler: TOmniOnTerminatedFunctionSimple):
  IOmniTaskControl; overload;
function OnTerminated(eventHandler: TOmniTaskTerminatedEvent):
  IOmniTaskControl; overload;
```

The termination handler can receive a reference to the controller of the task being terminated or it can be a parameterless procedure.

The termination handler is called in the context of the thread that created the task.

The example below shows how to write a termination handler to clear the task controller.

```
procedure NullTask(const task: IOmniTask);
begin
end;

var
  FTaskControl: IOmniTaskControl;

FTaskControl := CreateTask(NullTask)
FTaskControl.OnTerminated(
  procedure
  begin
    ShowMessage('Background task has terminated');
    FTaskControl := nil;
  end);
FTaskControl.Run;
```

Tasks running in a thread pool can also be terminated by calling the thread pool's `Cancel` or `CancelAll` functions which call `Terminate` internally.

4.10.5 ExitCode

A task can send a result to the controller by calling the `SetExitStatus` procedure. The task controller can access this result through the `ExitCode` and `ExitMessage` properties.

```
property ExitCode: integer read GetExitCode;
property ExitMessage: string read GetExitMessage;
```

By default, `ExitCode` contains value 0 and `ExitMessage` contains an empty string.

An application can use exit codes from 0 to $7FFFFFFF. Exit codes from $80000000 to $FFFFFFFF are reserved for internal OmniThreadLibrary use. Following exit codes (defined in the OtlCommon unit) have reserved meaning:

```
const
  // reserved exit statuses
  EXIT_OK                          = 0;
  EXIT_INTERNAL                    = integer($80000000);
  EXIT_THREADPOOL_QUEUE_TOO_LONG = EXIT_INTERNAL + 0;
  EXIT_THREADPOOL_STALE_TASK     = EXIT_INTERNAL + 1;
  EXIT_THREADPOOL_CANCELLED      = EXIT_INTERNAL + 2;
  EXIT_THREADPOOL_INTERNAL_ERROR = EXIT_INTERNAL + 3;
```

`EXIT_THREADPOOL_*` exit codes are described in the Thread pooling section.

4.10.6 Exceptions

If an unhandled exception is raised in the background task, it is caught by the OmniThreadLibrary code and stored in the `FatalException` property. When that happens, the background task terminates execution in a normal way (termination handler is called etc.).

The task controller can check the `FatalException` property to see whether an unhandled exception was raised (if not, the property will contain `nil`).

When the task controller is destroyed, so is the exception object. If you need to preserve the exception object, you can detach it from the task controller by calling the `DetachException` function.

```
function  DetachException: Exception;
property FatalException: Exception read GetFatalException;
```

See also demo `13_Exceptions`.

4.10.7 Sending messages to a task

The Communication subsystem section explains how the communication subsystem works and what methods can be used in your program. This part of the book will show a practical example of sending messages to a simple (not `TOmniWorker`-based) task. This example is based on the `2_TwoWayHello` demo.

The task is created in a standard manner.

```
FHelloTask :=
  OmniEventMonitor1.Monitor(CreateTask(RunHello, 'Hello'))
    .SetParameter('Message', 'Initial message')
    .Run;
```

A message is sent to the task through the `Comm` property.

```
const
  MSG_CHANGE_MESSAGE = 1;

FHelloTask.Comm.Send(MSG_CHANGE_MESSAGE, 'New message');
```

The task (`RunHello` procedure) must implement a loop which will wait on two events – `task.TerminateEvent` (to know when to stop) and `task.Comm.NewMessageEvent` (to know when a new message arrived). When the latter event is signalled, message can be read from the message queue.

You should always read and process all waiting messages, not just one.

```
procedure RunHello(const task: IOmniTask);
var
  msg    : string;
  msgData: TOmniValue;
  msgID : word;
begin
  msg := task.Param['Message'];
  repeat
    case DSiWaitForTwoObjects(task.TerminateEvent, task.Comm.NewMessageEvent,
           false, task.Param['Delay'])
    of
      WAIT_OBJECT_1:
        begin
          while task.Comm.Receive(msgID, msgData) do begin
            if msgID = MSG_CHANGE_MESSAGE then
              msg := msgData;
          end;
        end;
      WAIT_TIMEOUT:
        task.Comm.Send(0, msg);
      else
```

```
22          break; //repeat
23      end;
24    until false;
25  end;
```

This code uses `DSiWaitForTwoObjects` to wait on two events. That function is just a thin wrapper around Windows' `WaitForMultipleObjects`[5].

4.10.8 Receiving messages from a task

Messages sent from a task can be received and processed in many different ways. They can all be used with both simple and `TOmniWorker` tasks.

First approach is to use an event monitor to process messages.

Another way is to call an `OnMessage` method providing an event handler which can be a method or an anonymous method.

```
type
  TOmniTaskMessageEvent = procedure(const task: IOmniTaskControl;
    const msg: TOmniMessage) of object;
  TOmniOnMessageFunction = reference to procedure(const task: IOmniTaskControl;
    const msg: TOmniMessage);

function OnMessage(eventHandler: TOmniTaskMessageEvent): IOmniTaskControl; overload;
function OnMessage(eventHandler: TOmniOnMessageFunction): IOmniTaskControl; overload;
```

```
task := CreateTask(MyTask).OnMessage(
  procedure (const task: IOmniTaskControl; const msg: TOmniMessage)
  begin
    ShowMessage('Received message with ID ' + IntToStr(msg.MsgID));
  end).Run;
```

You can also set multiple message handlers, each handling a specific message ID.

```
type
  TOmniTaskMessageEvent = procedure(const task: IOmniTaskControl;
    const msg: TOmniMessage) of object;
  TOmniOnMessageFunction = reference to procedure(const task: IOmniTaskControl;
    const msg: TOmniMessage);

function OnMessage(msgID: word; eventHandler: TOmniTaskMessageEvent):
  IOmniTaskControl; overload;
function OnMessage(msgID: word; eventHandler: TOmniOnMessageFunction):
  IOmniTaskControl; overload;
```

```
task := CreateTask(MyTask);
task.OnMessage(
  MSG_FIRST,
  procedure (const task: IOmniTaskControl; const msg: TOmniMessage)
  begin
    ShowMessage('Received MSG_FIRST');
  end);
task.OnMessage(
  MSG_SECOND,
  procedure (const task: IOmniTaskControl; const msg: TOmniMessage)
  begin
    ShowMessage('Received MSG_SECOND');
  end);
task.Run;
```

The OmniThreadLibrary allows you to combine both approaches, providing specific message handlers for some messages and generic message handlers for all the rest.

```
task := CreateTask(MyTask);
task.OnMessage(MSG_FIRST, HandleFirstMessage);
task.OnMessage(MSG_SECOND, HandleSecondMessage);
task.OnMessage(HandleOtherMessages);
task.Run;
```

A message handler can also be provided through a `TOmniMessageExec` class, which is defined in the OtlTaskControl unit. This class can wrap any message handler type and is mainly intended for internal use.

```
function OnMessage(msgID: word; eventHandler: TOmniMessageExec):
  IOmniTaskControl; overload;
```

The last possibility is to dispatch all messages to an object which implements Delphi's `Dispatch` mechanism, for example to a form.

```
function OnMessage(eventDispatcher: TObject): IOmniTaskControl; overload;
```

An object should then define message handlers by using the Delphi's `message` procedures. Each should accept a `var TOmniMessage` parameter.

```
const
  WM_FIRST_MSG = WM_USER;
  WM_SECOND_MSG = WM_USER + 1;

type
  TForm1 = class(TForm)
  public
    FTask: IOmniTaskControl;
    procedure StartTask;
    procedure HandleFirstMessage(var msg: TOmniMessage); message WM_FIRST_MSG;
    procedure HandleSecondMessage(var msg: TOmniMessage); message WM_SECOND_MSG;
  end;

procedure TForm1.StartTask;
begin
  FTask := CreateTask(MyTask).OnMessage(Self).Run;
end;
```

Messages sent to the main thread should have message ID larger or equal than `WM_USER`.

4.10.9 ChainTo

Tasks can be chained together so that next task is started when the previous terminates. This is achieved by the `ChainTo` method.

```
function ChainTo(const task: IOmniTaskControl;
  ignoreErrors: boolean = false): IOmniTaskControl;
```

The code below starts the `task1` task. When it is finished, OmniThreadLibrary immediately starts `task2`. Note that in this case you should not start the second task explicitly.

```
task2 := CreateTask(SecondTask);
task1 := CreateTask(FirstTask).ChainTo(task2).Run;
```

If `ignoreErrors` is left at default (`False`), second task is only started if first task's `ExitCode` is `EXIT_OK` (0). If `ignoreErrors` is set to `True`, second task is started even if `ExitCode` of the first task is not 0.

4.10.10 Join / Leave

Calling `Join` adds the task to a task group. Calling `Leave` removes the task from the group.

```
1 function Join(const group: IOmniTaskGroup): IOmniTaskControl;
2 function Leave(const group: IOmniTaskGroup): IOmniTaskControl;
```

4.10.11 MonitorWith / RemoveMonitor

Calling `MonitorWith` attaches the task to an event monitor. Calling `RemoveMonitor` removes the task from the monitor.

```
1 function MonitorWith(const monitor: IOmniTaskControlMonitor): IOmniTaskControl;
2 function RemoveMonitor: IOmniTaskControl;
```

Calling `task.MonitorWith(otlMonitor)` is equivalent to calling `otlMonitor.Monitor(task)`. Calling `task.RemoveMonitor` is equivalent to calling `otlMonitor.Detach(task)`.

A task is automatically removed from the associated monitor when the task is destroyed.

Following three monitor events are useful for monitoring tasks.

```
 1 type
 2   TOmniMonitorTaskEvent = procedure(const task: IOmniTaskControl) of object;
 3   TOmniMonitorTaskMessageEvent = procedure(const task: IOmniTaskControl;
 4     const msg: TOmniMessage) of object;
 5
 6   TOmniEventMonitor = class(TComponent,
 7                             IOmniTaskControlMonitor,
 8                             IOmniThreadPoolMonitor)
 9   public
10     property OnTaskMessage: TOmniMonitorTaskMessageEvent;
11     property OnTaskTerminated: TOmniMonitorTaskEvent;
12     property OnTaskUndeliveredMessage: TOmniMonitorTaskMessageEvent;
13   end;
```

- `OnTaskMessage`

  This event is called whenever a message from the task is received. The `msg` parameter contains the message.

- `OnTaskTerminated`

  This event is called when a task terminates.

- `OnTaskUndeliveredMessage`

  If a task is terminated with unprocessed messages in its input queue, this event is called for each message.

`OnTaskMessage` is called for messages sent from the task. `OnTaskUndeliveredMessage` is called for messages sent to the task.

4.10.12 Enforced

The `Enforced` method regulates behaviour in a very specific situation – when a task is terminated before it even starts execution.

```
1 function Enforced(forceExecution: boolean = true): IOmniTaskControl;
```

In the default scenario, task is always started when the `Run` is called, even if it was terminated before that. While the task would typically immediately exit, it can still do some

processing.

```
procedure ShortLivedTask(const task: IOmniTask);
begin
  // this line will be executed
end;

task := CreateTask(ShortLivedTask);
task.Terminate;
task.Run;
```

If `Enforced(False)` is called, a terminated process won't be started at all.

```
procedure UnstartedTask(const task: IOmniTask);
begin
  // this line will never be executed
end;

task := CreateTask(UnstartedTask);
task.Enforced(false);
task.Terminate;
task.Run;
```

This method also regulates behaviour of tasks that were scheduled to the thread pool.

4.10.13 Unobserved

Calling `Unobserved` adds an implicit owner to the task so that you don't have to store returned `IOmniTaskController` interface in a variable or a field.

```
function Unobserved: IOmniTaskControl;
```

See the Task controller needs an owner section for an example.

4.10.14 Cancellation token / CancelWith

OmniThreadLibrary implements the concept of a cancellation token, which is a synchronisation mechanism that allows multiple tasks to be cancelled with a single command.

Cancellation must be cooperative, i.e. the task must watch its `CancellationToken` property and exit when it becomes signalled.

```
function CancelWith(const token: IOmniCancellationToken): IOmniTaskControl;

property CancellationToken: IOmniCancellationToken read GetCancellationToken;
```

When using `TOmniWorker` tasks, the `TerminateWhen` mechanism may be a better solution.

4.10.15 Lock / WithLock

Sometimes you have to establish a common synchronisation primitive between the main program and one (or more) background tasks. In such case, the `WithLock` method can be called to assign a synchronisation primitive (such as critical section) to the task's `Lock` property.

```
function WithLock(const lock: TSynchroObject;
  autoDestroyLock: boolean = true): IOmniTaskControl; overload;
function WithLock(const lock: IOmniCriticalSection): IOmniTaskControl; overload;

property Lock: TSynchroObject read GetLock;
```

If you pass in a `TSynchroObject` descendant, the default behaviour is to automatically destroy this object when the task is destroyed. In case you are passing the same synchronisation object to two or more tasks, this is not a good idea and second `WithLock` parameter should be set to `False`. Alternatively, you can use an `IOmniCriticalSection` which will destroy itself automatically when it is no longer used.

See also demo `12_Lock`.

4.10.16 WithCounter

Similar to the `Lock` mechanism, `WithCounter` connects a task with a thread-safe counter, `IOmniCounter`.

```
1 function WithCounter(const counter: IOmniCounter): IOmniTaskControl;
```

See also demo `14_TerminateWhen`.

4.10.17 SetPriority

By calling `SetPriority` you can specify the priority of the thread that will execute the task. By default the priority is set to `tpNormal` which corresponds to Windows priority level `THREAD_PRIORITY_NORMAL`.

```
1 type
2   TOTLThreadPriority = (tpIdle, tpLowest, tpBelowNormal, tpNormal,
3     tpAboveNormal, tpHighest);
4
5 function SetPriority(threadPriority: TOTLThreadPriority): IOmniTaskControl;
```

4.10.18 SetQueueSize

The `SetQueueSize` sets the length for message queues. By default, message queue length is set to `1000` messages.

```
1 function  SetQueueSize(numMessages: integer): IOmniTaskControl;
```

This method must be called before the communication channel is used for the first time. The safest way is to call it before the task is run or scheduled.

4.11 TOmniWorker tasks

In the heart of every sufficiently complicated thread code there lies a complicated loop similar to the one below.

```
1 type
2   handles: array [0..3] of THandle;
3   waitRes: DWORD;
4
5 begin
6   handles[0] := TerminateEvent;
7   handles[1] := FWakeUp;
8   handles[2] := FResetFilters;
9   handles[3] := FResetFullStream;
10
11  repeat
12    waitRes := WaitForMultipleObjects(numHandles, 4, INFINITE, false);
13    case waitRes of:
14      WAIT_OBJECT_0:
15        Exit;
16      WAIT_OBJECT_0+1:
17        ProcessIncomingData;
18      WAIT_OBJECT_0+2:
```

```
19          ResetFilters;
20        WAIT_OBJECT_0+3:
21          ResetStream;
22        else Exit; //WAIT_TIMEOUT, WAIT_ABANDONED
23      end;
24    until false;
25  end;
```

You can write OmniThreadLibrary-based code in the same manner (see simple tasks) but the library also offers a much better way. Write the task as a descendant of the `TOmniWorker` class and OmniThreadLibrary will implement the main loop for you.

Every `TOmniWorker`-type task executes an internal loop which can be represented by the following pseudocode.

```
1  if not Initialize then
2    Exit;
3  repeat
4    waitRes := WaitForEvent;
5    if waitRes = TerminateEvent then
6      break
7    else
8      DispatchEvent(waitRes);
9  until false;
10 Cleanup;
```

Firstly, the `Initialize` function is called. In the `TOmniWorker` class it does nothing, just returns `True`. You can override it with your own function which does task-specific initialization. It can optionally return `False` which signifies that the task should not be executed at all.

For compatibility with future OmniThreadLibrary versions which may return `False` from the base `Initialize` under some conditions, it is recommended to write overridden `Initialize` in the following form:

```
1 function Initialize: boolean;
2 begin
3   Result := inherited Initialize;
4   if not Result then
5     Exit;// do the rest of initialization here
6 end;
```

Next, the wait and dispatch loop is run. It will watch for following events:

- `TerminateEvent`,
- Other termination events (`TerminateWhen`),
- New messages from the task controller,
- New messages from other tasks (`RegisterComm`),
- Windows synchronization objects (`RegisterWaitObject`),
- Timers (`SetTimer`),
- Windows messages (`MsgWait`, `Alertable`).

In versions before [3.04] you could only register 57 termination events, additional communication channels, external events and timers (all together). This limitation is no longer present in release [3.04] and newer.

When an event occurs, an appropriate action is executed. For example, messages are dispatched (using the Delphi's `Dispatch` mechanism) to the task so you can use `message` methods for message processing (see Receiving messages below).

When a `TerminateEvent` is signalled (which happens when controller's `Terminate` method is

called) or when one of events passed in the `TerminateWhen` method is signalled, the task's main loop exits. The `Cleanup` method is then called. In the base class, `Cleanup` is implemented as an empty method; override it to implement the task-specific clean-up.

`TOmniWorker` tasks are not really suitable for single shot events. Use simple tasks instead.

4.11.1 WaitForInit

If the owner of the task wants to wait until the `Initialize` method has completed its work, it can call the `WaitForInit` method.

```
1 function WaitForInit: boolean;
```

4.11.2 Task

The `TOmniWorker` class implements a public property `Task` returning the task's management interface `IOmniTask`. You can use it to access any task-specific property, such as the communication interface (`Comm`).

```
1 property Task: IOmniTask read GetTask write SetTask;
```

This property is available in the `Initialize` and `Cleanup` methods but not in the constructor and destructor.

4.11.3 Receiving messages

Messages sent from the controller or from other tasks are passed to the Delphi's `Dispatch` mechanism. For each message that we want to process we must therefore write a `message` method.

To demonstrate this approach, here is a short example extracted from the `5_TwoWayHello_without_loop` demo.

The task is implemented as an instance of the `TAsyncHello` class which is derived from the `TOmniWorker`. The task will do some initialization in the `Initialize` method. It will then handle two messages - `MSG_CHANGE_MESSAGE` and `MSG_SEND_MESSAGE`.

When the `MSG_CHANGE_MESSAGE` is received, worker will change the internal message text which was initially assigned in `Initialize`. When the `MSG_SEND_MESSAGE` is received, worker will send this message back to the owner.

```
1  const
2    MSG_CHANGE_MESSAGE = 1;
3    MSG_SEND_MESSAGE   = 2;
4
5  type
6    TAsyncHello = class(TOmniWorker)
7    strict private
8      aiMessage: string;
9    public
10     function  Initialize: boolean; override;
11     procedure OMChangeMessage(var msg: TOmniMessage);
12       message MSG_CHANGE_MESSAGE;
13     procedure OMSendMessage(var msg: TOmniMessage);
14       message MSG_SEND_MESSAGE;
15   end;
```

We can write those two message handlers identical to the way we write Windows message handlers. Message IDs can take any value from 0 to $FFFE. Message ID $FFFF is reserved

for internal purposes.

```
function TAsyncHello.Initialize: boolean;
begin
  aiMessage := Task.ParamByName['Message'];
  Result := true;
end;

procedure TAsyncHello.OMChangeMessage(var msg: TOmniMessage);
begin
  aiMessage := msg.MsgData;
end;

procedure TAsyncHello.OMSendMessage(var msg: TOmniMessage);
begin
  Task.Comm.Send(0, aiMessage);
end;
```

4.11.4 RegisterComm

You can send and receive messages directly from one task to another. To do that, you should firstly create a two-way communication channel and pass each endpoint of this channel to one of the tasks. A task should then call `RegisterComm` function to start listening on an endpoint. To stop listening, call the `UnregisterComm` function.

Received messages are dispatched in the same way as messages received from the task controller.

```
procedure RegisterComm(const comm: IOmniCommunicationEndpoint);
procedure UnregisterComm(const comm: IOmniCommunicationEndpoint);
```

The `8_RegisterComm` demo demonstrates this.

A two-way channel (`FCommChannel`) is created with space for 1024 messages. Each of its endpoints is passed to one instance of the `TCommTester` class and each instance is started as its own task.

```
procedure TfrmTestRegisterComm.FormCreate(Sender: TObject);
begin
  FCommChannel := CreateTwoWayChannel(1024);
  FClient1 := CreateTask(TCommTester.Create(FCommChannel.Endpoint1))
    .MonitorWith(OmniTED)
    .Run;
  FClient2 := CreateTask(TCommTester.Create(FCommChannel.Endpoint2))
    .MonitorWith(OmniTED)
    .Run;
end;
```

The `TCommTester` constructor just stores this endpoint in an internal field. It is then used in the overridden `Initialize` as a parameter to the `RegisterComm` call. [We cannot call `RegisterComm` in the constructor as the `Task` property is not available at that time yet.]

```
type
  TCommTester = class(TOmniWorker)
  strict private
    ctComm: IOmniCommunicationEndpoint;
  public
    constructor Create(commEndpoint: IOmniCommunicationEndpoint);
    function  Initialize: boolean; override;
    procedure OMForward(var msg: TOmniMessage); message MSG_FORWARD;
    procedure OMForwarding(var msg: TOmniMessage); message MSG_FORWARDING;
  end;

constructor TCommTester.Create(commEndpoint: IOmniCommunicationEndpoint);
begin
```

```
14    inherited Create;
15    ctComm := commEndpoint;
16  end;
17
18  function TCommTester.Initialize: boolean;
19  begin
20    Result := inherited Initialize;
21    if Result then
22      Task.RegisterComm(ctComm);
23  end;
```

To send a message to another task, use the `Send` method of the `IOmniCommunicationEndpoint`.

```
1  procedure TCommTester.OMForward(var msg: TOmniMessage);
2  begin
3    Task.Comm.Send(MSG_NOTIFY_FORWARD, msg.MsgData);
4    ctComm.Send(MSG_FORWARDING, msg.MsgData);
5  end;
6
7  procedure TCommTester.OMForwarding(var msg: TOmniMessage);
8  begin
9    Task.Comm.Send(MSG_NOTIFY_RECEPTION, msg.MsgData);
10 end;
```

4.11.5 Invoke

Instead of sending messages that are processed in the task you can also instruct the task to execute a specific code. OmniThreadLibrary provides three variations of the same mechanism – `Invoke`. They allow you to call a method by providing its name, call a method by providing its address or to execute an anonymous method.

```
1  function Invoke(const msgName: string): IOmniTaskControl; overload;
2  function Invoke(const msgName: string; msgData: array of const):
3    IOmniTaskControl; overload;
4  function Invoke(const msgName: string; msgData: TOmniValue):
5    IOmniTaskControl; overload;
6
7  function Invoke(const msgMethod: pointer): IOmniTaskControl; overload;
8  function Invoke(const msgMethod: pointer; msgData: array of const):
9    IOmniTaskControl; overload;
10 function Invoke(const msgMethod: pointer; msgData: TOmniValue): IOmniTaskControl; overload;
11
12 function Invoke(remoteFunc: TOmniTaskControlInvokeFunction): IOmniTaskControl; overload;
13 function Invoke(remoteFunc: TOmniTaskControlInvokeFunctionEx): IOmniTaskControl; overload;
```

`Invoke` uses the communication system to execute the code inside the task. It sends a special internal message (with the reserved ID $FFFF) to the task. Internal `TOmniWorker` code catches this message and instead of dispatching it to your task object the code referenced from the message is executed. The code is therefore not executed when you call the `Invoke` but some undeterminable time later.

You can pass some data to an invoked method. In `Invoke` you can add a second parameter which can be a `TOmniValue` or an array of elements which is converted to a `TOmniValue`. To receive this data, the method on the task side must either accept a `const TOmniValue` parameter or a `var TObject` parameter. In the latter case you must, of course, provide an object (of any class) as the second `Invoke` parameter.

Invoked methods must be marked `public` or `published`.

Let's see a simple example. We have a very simple worker with three public methods.

```
1  type
2    TMyObj = class
3      Val: integer;
```

```
4     constructor Create(value: integer);
5   end;
6
7   TMyTask = class(TOmniWorker)
8   public
9     procedure Test1;
10    procedure Test2(const value: TOmniValue);
11    procedure Test3(var obj: TObject);
12  end;
13
14 { TMyObj }
15
16 constructor TMyObj.Create(value: integer);
17 begin
18   Val := value;
19 end;
20
21 { TMyTask }
22
23 procedure TMyTask.Test1;
24 begin
25 end;
26
27 procedure TMyTask.Test2(const value: TOmniValue);
28 begin
29 end;
30
31 procedure TMyTask.Test3(var obj: TObject);
32 begin
33   obj.Free;
34 end;
```

You can now call these three methods in the following manner.

```
1 FTask := CreateTask(TMyTask.Create()).Run;
2 FTask.Invoke('Test1');
3 FTask.Invoke('Test2', [1, 2, 3]);
4 FTask.Invoke('Test3', TMyObj.Create(42));
5
6 //This would cause a runtime error because method 'Test4' doesn't exist.
7 //FTask.Invoke('Test4');
```

Alternatively, you can provide an address of the method instead of its name which gives you some compile-time checking – if the code tries to call an inexistent method it will not compile.

```
1 FTask := CreateTask(TMyTask.Create()).Run;
2 FTask.Invoke(@TMyTask.Test1);
3 FTask.Invoke(@TMyTask.Test2, [1, 2, 3]);
4 FTask.Invoke(@TMyTask.Test3, TMyObj.Create(42));
5
6 //This would not compile because method 'Test4' doesn't exist.
7 //FTask.Invoke(@TMyTask.Test4);
```

The last two versions of `Invoke` allow you to execute an anonymous method in the background task. One will run a parameterless method while another executes a method accepting an `IOmniTask` parameter which you can use to control the task.

```
1 FTask := CreateTask(TMyTask.Create()).Run;
2 FTask.Invoke(
3   procedure
4   begin
5     // some code
6   end);
7 FTask.Invoke(
8   procedure (const task: IOmniTask)
9   begin
10    // some code
```

```
11    end);
```

See also demo 43_InvokeAnonymous.

4.11.6 Windows message & APC processing

The internal main loop in the TOmniWorker can optionally process and dispatch Windows messages. This is especially important if your task creates components that use Windows messages for normal operation.

To force Windows message processing, you must call the MsgWait method of the task controller. You can optionally provide a wake mask.

```
1 function MsgWait(wakeMask: DWORD = QS_ALLEVENTS): IOmniTaskControl;
```

If your background code uses asynchronous procedure calls (for example if it is reading from a file asynchronously), you should call the Alertable method which will enable the MWMO_ALERTABLE flag.

```
1 function Alertable: IOmniTaskControl;
```

More information about the wake mask and about the alertable flag can be found in the MSDN.

4.11.7 Timers

The TOmniWorker model greatly simplifies executing repeated tasks inside the task worker. You can set up multiple timers and associate them with the code in few different ways.

```
1 function ClearTimer(timerID: integer): IOmniTaskControl;
2 function SetTimer(timerID: integer; interval_ms: cardinal;
3   const timerMessage: TOmniMessageID): IOmniTaskControl;
4 procedure SetTimer(timerID: integer; interval_ms: cardinal;
5   const timerMessage: TProc); overload;
6 procedure SetTimer(timerID: integer; interval_ms: cardinal;
7   const timerMessage: TProc<integer>); overload;
```

To set up a timer, call the SetTimer function. It accepts three parameters – timer ID, interval (in milliseconds) and a timer message. The latter is a parameter specifying the code to be executed every interval_ms milliseconds and can be specified in three different ways which are discussed below.

Two other SetTimer overloads, introduced in version [3.07.3] allow you to execute an anonymous function as a timer event. The latter will pass timer ID to the anonymous function's integer parameter.

To clear a timer (so that the associated code is not called anymore) call the ClearTimer function.

Don't expect that timers will be called with a millisecond precision. Windows OS doesn't support that.

Two additional SetTimer overloads (not shown in the book) are provided only for backward compatibility with very old code and should not be used in new programs.

The code below shows four ways in which a timer method can be specified.

- A name of the timer method (Timer1). The method must have at least public visibility.

The method can optionally accept one parameter of type `TOmniValue` which will contain the timer ID.

- An address of the timer method (`@TMyTask.Timer2`). The method can optionally accept one parameter of type `TOmniValue` which will contain the timer ID.
- A message ID (`MSG_TIMER3`). This message ID must not conflict with other messages processed in the background worker. The message handler (`Timer3` in the example above) must accept a `TOmniMessage` parameter. Its `MsgID` field will contain the message ID (`MSG_TIMER3`) and its `MsgData` field will contain the timer ID (`3`).
- An anonymous method. [3.07.3] It can have zero parameters or one integer parameter, which will receive the timer ID.

```
const
  MSG_TIMER3 = 1;

type
  TMyTask = class(TOmniWorker)
  public
    procedure Timer1;
    procedure Timer2(const value: TOmniValue);
    procedure Timer3(var msg: TOmniMessage); message MSG_TIMER3;
  end;

{ TMyTask }

procedure TMyTask.Timer1;
begin
  // called every 150 ms
end;

procedure TMyTask.Timer2(const value: TOmniValue);
begin
  // called every 200 ms
  // value contains the timer ID
end;

procedure TMyTask.Timer3(var msg: TOmniMessage);
begin
  // called every 250 ms
  // msg.MsgData contains the timer ID
end;

FTask := CreateTask(TMyTask.Create()).Run;
FTask.SetTimer(1, 150, 'Timer1');
FTask.SetTimer(2, 200, @TMyTask.Timer2);
FTask.SetTimer(3, 250, MSG_TIMER3);
FTask.SetTimer(4, 333,
  procedure (timerID: integer)
  begin
    // called every 333 seconds
    // timerID contains the timer ID
  end);
```

Timers can also be set and cleared from inside the task.

4.11.8 TerminateWhen

By using the `TerminateWhen` function you can set up a termination trigger – a signal that will stop the background task execution. This can be either a Windows synchronization primitive (such as event) or a cancellation token.

```
function TerminateWhen(event: THandle): IOmniTaskControl; overload;
function TerminateWhen(token: IOmniCancellationToken): IOmniTaskControl; overload;
```

The use of the `TerminateWhen` function is demonstrated in the `14_TerminateWhen` demo.

4.11.9 UserData

Sometimes you would want to associate some data with the task controller. The `UserData` property can be used for this purpose. This data is never accessed from the OmniThreadLibrary code; it is only managed by the library for your convenience. `UserData` is available via the `IOmniTaskController` interface and cannot be used in the background task.

This functionality may be for example be used to set some data when the task is created and then access this data in the termination handler.

```
function SetUserData(const idxData: TOmniValue;
  const value: TOmniValue): IOmniTaskControl;
property UserData[const idxData: TOmniValue]: TOmniValue
  read GetUserDataVal write SetUserDataVal;
```

Data can be accessed through an integer index or through a name. Both lines below are valid.

```
taskController.UserData[0] := myTaskList.Add(taskController);
taskController.UserData['token'] := 'something';
```

4.12 Task groups

Task group is a mechanism that allows you to treat a number of tasks as one. You can start all tasks, send a message to all tasks in a group, terminate all tasks at once and wait for all to terminate.

See also demo `15_TaskGroup`.

```
type
  IOmniTaskGroup = interface
    ['{B36C08B4-0F71-422C-8613-63C4D04676B7}']
    function  GetTasks: IOmniTaskControlList;
  //
    function  Add(const taskControl: IOmniTaskControl): IOmniTaskGroup;
    function  GetEnumerator: IOmniTaskControlListEnumerator;
    function  RegisterAllCommWith(const task: IOmniTask): IOmniTaskGroup;
    function  Remove(const taskControl: IOmniTaskControl): IOmniTaskGroup;
    function  RunAll: IOmniTaskGroup;
    procedure SendToAll(const msg: TOmniMessage);
    function  TerminateAll(maxWait_ms: cardinal = INFINITE): boolean;
    function  UnregisterAllCommFrom(const task: IOmniTask): IOmniTaskGroup;
    function  WaitForAll(maxWait_ms: cardinal = INFINITE): boolean;
    property  Tasks: IOmniTaskControlList read GetTasks;
  end;

  TOmniTaskGroup = class(TInterfacedObject, IOmniTaskGroup)
  public
    constructor Create;
    destructor  Destroy; override;
    function  Add(const taskControl: IOmniTaskControl): IOmniTaskGroup;
    function  GetEnumerator: IOmniTaskControlListEnumerator;
    function  RegisterAllCommWith(const task: IOmniTask): IOmniTaskGroup;
    function  Remove(const taskControl: IOmniTaskControl): IOmniTaskGroup;
    function  RunAll: IOmniTaskGroup;
    procedure SendToAll(const msg: TOmniMessage);
    function  TerminateAll(maxWait_ms: cardinal = INFINITE): boolean;
    function  UnregisterAllCommFrom(const task: IOmniTask): IOmniTaskGroup;
    function  WaitForAll(maxWait_ms: cardinal = INFINITE): boolean;
    property Tasks: IOmniTaskControlList read GetTasks;
  end;

function CreateTaskGroup: IOmniTaskGroup;
```

- `Add`

  Adds a task to the group. See also `Join`.

- `GetEnumerator`

  Allows you to use the task group in a `for..in` statement returning all tasks in the group.

- `RegisterAllCommWith`

  Registers communication channel of all tasks in the group with another task (the parameter to the `RegisterAllCommWith` function) so they can all send messages to that task.

- `Remove`

  Removes a task from the group. See also `Leave`.

- `RunAll`

  Starts all tasks in the group.

- `SendToAll`

  Sends the message to every task in the group.

- `TerminateAll`

  Terminates all tasks, waiting up to `maxWait_ms` milliseconds for all of them to complete. The function returns `True` if all tasks are terminated when the function returns.

- `UnregisterAllCommFrom`

  Unregisters additional communication channel from all tasks in the group.

- `WaitForAll`

  Waits up to `maxWait_ms` milliseconds for all tasks to complete. The function returns `True` if all tasks are terminated when the function returns.

- `Tasks`

  Provides access to the list containing all tasks in the group.

Starting with [3.04], task groups are not limited in size. In previous releases, a task group could contain at most 60 tasks.

4.13 IOmniTask interface

The background worker can use the `IOmniTask` interface (defined in the OtlTask unit) to communicate with the task controller and to control the task's execution. The worker can access the interface in two different ways. For the simple tasks the interface is passed as a parameter to the task worker method. `TOmniWorker` tasks can access it through the `Task` property.

The `IOmniTask` interface exposes following methods.

```
1 type
2   TOmniTaskInvokeFunction = reference to procedure;
3
4   IOmniTask = interface
5     procedure ClearTimer(timerID: integer = 0);
```

```
    procedure Enforced(forceExecution: boolean = true);
    procedure Invoke(remoteFunc: TOmniTaskInvokeFunction);
    procedure InvokeOnSelf(remoteFunc: TOmniTaskInvokeFunction);
    procedure RegisterComm(const comm: IOmniCommunicationEndpoint);
    procedure RegisterWaitObject(waitObject: THandle;
      responseHandler: TOmniWaitObjectMethod);
    procedure SetException(exceptionObject: pointer);
    procedure SetExitStatus(exitCode: integer; const exitMessage: string);
    procedure SetProcessorGroup(procGroupNumber: integer);
    procedure SetNUMANode(numaNodeNumber: integer);
    procedure SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TOmniMessageID);
    procedure SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TProc); overload;
    procedure SetTimer(timerID: integer; interval_ms: cardinal;
      const timerMessage: TProc<integer>); overload;
    procedure StopTimer;
    procedure Terminate;
    function  Terminated: boolean;
    function  Stopped: boolean;
    procedure UnregisterComm(const comm: IOmniCommunicationEndpoint);
    procedure UnregisterWaitObject(waitObject: THandle);
    property CancellationToken: IOmniCancellationToken read GetCancellationToken;
    property Comm: IOmniCommunicationEndpoint read GetComm;
    property Counter: IOmniCounter read GetCounter;
    property Implementor: TObject read GetImplementor;
    property Lock: TSynchroObject read GetLock;
    property Name: string read GetName;
    property Param: TOmniValueContainer read GetParam;
    property TerminateEvent: THandle read GetTerminateEvent;
    property ThreadData: IInterface read GetThreadData;
    property UniqueID: int64 read GetUniqueID;
  end; { IOmniTask }
```

4.13.1 Name and ID

The worker can access its name and unique ID through the `Name` and `UniqueID` properties.

```
property Name: string read GetName;
property UniqueID: int64 read GetUniqueID;
```

4.13.2 Parameters

To access the parameters provided by the task owner, the worker can use the `Param` property.

```
property Param: TOmniValueContainer read GetParam;
```

More information is provided in the Simple Tasks/Parameters section.

4.13.3 Termination

To terminate itself, a simple task should just exit from the worker method. A `TOmniWorker` task, on the other hand, should call the `Terminate` method of the `IOmniTask` interface.

```
procedure Terminate;
function  Terminated: boolean;
function  Stopped: boolean;
property TerminateEvent: THandle read GetTerminateEvent;
```

Other termination-related methods and properties are.

- `Terminated`

Returns `True` when a worker was requested to stop, either by itself or by the task controller.

- Stopped

Returns `True` when a worker has fully stopped. There's no use in calling this method from the worker itself as it will always return `False` while the worker is still active.

- TerminateEvent

Returns the event that is signalled when a task controller requires the worker to stop. The worker can wait on this event but should never signal it. `Terminate` should be called instead to terminate the worker.

4.13.4 Exit status

A task can send a result to the controller by calling the `SetExitStatus` procedure.

```
1 procedure SetExitStatus(exitCode: integer; const exitMessage: string);
```

The program can access this program through the task controller interface.

4.13.5 Exceptions

While exceptions in a background worker code are automatically caught, a task can also set an associated exception manually, by calling `SetException`.

```
1 procedure SetException(exceptionObject: pointer);
```

This exception is available to the main program through the `FatalException` property.

4.13.6 Communication

To send a message to the task controller (and through it to the main program), a background task can use the `Comm` property.

```
1 property Comm: IOmniCommunicationEndpoint read GetComm;
```

Any data can be packed in a `TOmniValue` record and passed to the `Comm.Send` method. The mechanism is the same as sending the data from the main program to the background worker.

Messages can be received in a different ways, depending on the worker type. See the appropriate sections of this manual for simple tasks and for `TOmniWorker` tasks.

A simple trick can be used to send a message from a task back to itself – instead of sending it on the `Comm` channel, you send it on the `Comm.OtherEndpoint` channel.

```
1 Task.Comm.OtherEndpoint.Send(MSG_FROM_TASK, 'Sent to Self');
```

A background task can use the communication channel to request asynchronous execution of code in the context of its owner (the thread which created the task).

```
1 type
2   TOmniTaskInvokeFunction = reference to procedure;
3
4 procedure Invoke(remoteFunc: TOmniTaskInvokeFunction);
```

This mechanism works by posting a special message containing the function to be executed from the task back to the task controller. OmniThreadLibrary catches that special message and instead of dispatching it to the normal message processing code, it will execute the attached function.

For this to work, somebody has to process received messages. This happens automatically with high-level abstractions and with low-level code that uses the `OnMessage` message dispatch.

If you write a simple low-level task that doesn't include a call to `OnMessage`, then the function will only be executed when the task is destroyed. To fix this problem, execute `OnMessage(Self)` on the task controller interface.

Example: `FTask := CreateTask(TMyTask.Create(), 'worker').OnMessage(Self).Run;`

Since version [3.07.3], `IOmniTask` interface implements method `InvokeOnSelf`, which works the same as `Invoke`, except that it posts a message back to the task itself. In other words, `InvokeOnSelf` puts an anonymous method into the task's message queue while `Invoke` puts it into the task controller's message queue. `InvokeOnSelf` will only work correctly when the task is implemented as a `TOmniWorker` task.

```
type
  TOmniTaskInvokeFunction = reference to procedure;

procedure InvokeOnSelf(remoteFunc: TOmniTaskInvokeFunction);
```

A `TOmniWorker` task can register additional communication channels to enable automatic processing of messages sent across those channels. This mechanism is described in the `TOmniWorker` tasks section.

```
procedure RegisterComm(const comm: IOmniCommunicationEndpoint);
procedure UnregisterComm(const comm: IOmniCommunicationEndpoint);
```

4.13.7 Timers

A `TOmniWorker` task can use timers to automate repeating tasks. This functionality is explained in the `TOmniWorker` tasks section.

```
procedure ClearTimer(timerID: integer = 0);
procedure SetTimer(timerID: integer; interval_ms: cardinal;
  const timerMessage: TOmniMessageID); overload;
procedure SetTimer(timerID: integer; interval_ms: cardinal;
  const timerMessage: TProc); overload;
procedure SetTimer(timerID: integer; interval_ms: cardinal;
  const timerMessage: TProc<integer>); overload;
```

4.13.8 RegisterWaitObject

A `TOmniWorker` task can register external synchronization object (typically an event) to be included in the main loop. An associated method will be called when a synchronization object becomes signalled.

```
type
  TOmniWaitObjectMethod = procedure of object;

procedure RegisterWaitObject(waitObject: THandle;
  responseHandler: TOmniWaitObjectMethod);
procedure UnregisterWaitObject(waitObject: THandle);
```

Use `RegisterWaitObject` to register external synchronization object with the main loop and `UnregisterWaitObject` to remove the synchronization object from the main loop.

4.13.9 CancellationToken

If a cancellation token is associated with the background task, it can be accessed through

the `CancellationToken` property.

```
1 property CancellationToken: IOmniCancellationToken read GetCancellationToken;
```

4.13.10 Lock

If a lock is associated with the background task, it can be accessed through the `Lock` property.

```
1 property Lock: TSynchroObject read GetLock;
```

4.13.11 Counter

If a threadsafe counter is associated with the background task, it can be access through the `Counter` property.

```
1 property Counter: IOmniCounter read GetCounter;
```

4.13.12 Processor groups and NUMA nodes

On a system with multiple processor groups you can use `SetProcessorGroup` [3.06] function to specify a processor group this task should run on.

On a system with multiple NUMA nodes you can use `SetNUMANode` [3.06] function to specify a NUMA node this task should run on.

An information about existing processor groups and NUMA nodes can be accessed through the `Environment` object.

This information should be in most cases managed via `IOmniTaskControl.ProcessorGroup`, `IOmniTaskControl.NUMANode`, `IOmniThreadPool.ProcessorGroups`, and `IOmniThreadPool.NUMANodes`.

4.13.13 Internal and obsolete functions

The `Enforced` method works the same as the `IOmniTaskControl` equivalent. As it must be called before the task starts execution, it cannot be used from the background task itself. OmniThreadLibrary uses this function in the thread pool implementation.

```
1 procedure Enforced(forceExecution: boolean = true);
```

The `Implementor` object returns an object implementing the `IOmniTask` interface.

```
1 property Implementor: TObject read GetImplementor;
```

The `ThreadData` property is used to access the internal interface used in the thread pool's `SetThreadDataFactory` mechanism.

```
1 property ThreadData: IInterface read GetThreadData;
```

The Building a connection pool example contains more information on this topic.

The `StopTimer` method is obsolete and should not be used anymore. Use the `ClearTimer` instead.

```
1 procedure StopTimer;
```

5. Synchronization

Although the OmniThreadLibrary treats communication as a superior approach to locking, there are still times when using "standard" synchronization primitives such as a critical section are unavoidable. As the standard Delphi/Windows approach to locking is very low-level, OmniThreadLibrary builds on it and improves it in some significant ways. All these improvements are collected in the OtlSync unit and are described in the following sections. The only exception is the waitable value class/interface, which is declared in the OtlCommon unit.

This part of the book assumes that you have a basic understanding of locking. If you are new to the topic, you should first read the appropriate chapters from one of the books mentioned in the introduction.

5.1 Critical sections

The most useful synchronisation primitive for multithreaded programming is indubitably the critical section[67]

OmniThreadLibrary simplifies sharing critical sections between a task owner and a task with the use of the `WithLock` method. High-level tasks can access this method through the task configuration block.

I was always holding the opinion that locks should be as granular as possible. Putting many small locks around many unrelated pieces of code is better than using one giant lock for everything. However, I find that programmers frequently use one or few locks because managing many critical sections can be a bother.

To help you with writing a better code, OmniThreadLibrary implements three extensions to the Delphi's `TCriticalSection` class - `IOmniCriticalSection`, `TOmniCS` and `Locked<T>`.

5.1.1 IOmniCriticalSection

Delphi implements critical section support with a `TCriticalSection` class which must be created and destroyed in the code. (There is also a `TRTLCriticalSection` record, but it is only supported on Windows.) OmniThreadLibrary extends this implementation with an `IOmniCriticalSection` interface, which you only have to create. The compiler will make sure that it is destroyed automatically at the appropriate place.

```
1 type
2   IOmniCriticalSection = interface
3     procedure Acquire;
4     procedure Release;
5     function  GetSyncObj: TSynchroObject;
6     property LockCount: integer read GetLockCount;
7   end;
8
9 function CreateOmniCriticalSection: IOmniCriticalSection;
```

`IOmniCriticalSection` uses `TCriticalSection` internally. It acts just as a proxy that calls `TCriticalSection` functions. Besides that, it provides an additional functionality by counting the number of times a critical section has been acquired, which can help a lot while debugging. This counter can be read through the `LockCount` property.

A critical section can be acquired multiple times from one thread. For example, the following code is perfectly valid:

```
cSec := CreateOmniCriticalSection; //LockCount = 0
cSec.Acquire; //LockCount = 1
cSec.Acquire; //LockCount = 2
cSec.Release; //LockCount = 1
cSec.Release; //LockCount = 0
```

Additionally, `IOmniCriticalSection` doesn't use `TCriticalSection` directly, but wraps it into a larger object as suggested by Eric Grange.

5.1.2 TOmniCS

Another `TCriticalSection` extension found in the OmniThreadLibrary is the `TOmniCS` record. It allows you to use a critical section by simply declaring a record in appropriate place.

Using `TOmniCS`, locking can be as simple as this:

```
uses
  GpLists,
  OtlSync;

procedure ProcessList(const intf: IGpIntegerList);
begin
  //...
end;

var
  lock: TOmniCS;
  intf: IGpIntegerList;

procedure Test1;
begin
  intf := TGpIntegerList.Create;
  //...
  lock.Acquire;
  try
    ProcessList(intf);
  finally lock.Release; end;
end;
```

`TOmniCS` is implemented as a record with one private field holding the `IOmniCriticalSection` interface.

```
type
  TOmniCS = record
  strict private
    ocsSync: IOmniCriticalSection;
  private
    function  GetLockCount: integer; inline;
    function  GetSyncObj: TSynchroObject; inline;
  public
    procedure Initialize;
    procedure Acquire; inline;
    procedure Release; inline;
    property LockCount: integer read GetLockCount;
    property SyncObj: TSynchroObject read GetSyncObj;
  end;
```

The `Release` method merely calls the `Release` method on the internal interface, while the `Acquire` method is more tricky as it has to initialize the `ocsSync` field first.

```
procedure TOmniCS.Acquire;
begin
  Initialize;
```

```
  ocsSync.Acquire;
end;

procedure TOmniCS.Release;
begin
  ocsSync.Release;
end;
```

The initialization uses a global critical section to synchronize access to the code that should not be executed from two threads at once.

```
procedure TOmniCS.Initialize;
begin
  if not assigned(ocsSync) then begin
    GOmniCSInitializer.Acquire;
    try
      if not assigned(ocsSync) then
        ocsSync := CreateOmniCriticalSection;
    finally GOmniCSInitializer.Release; end;
  end;
end;
```

5.1.3 Locked<T>

`TOmniCS` is a great simplification of the critical section concept, but it still requires you to declare a separate locking entity. If this locking entity is only used to synchronize access to a specific instance (being that an object, record, interface or even a simple type) it is often better to declare a variable/field of type `Locked<T>` which combines any type with a critical section.

Using `Locked<T>`, the example from the `TOmniCS` section can be rewritten as follows.

```
uses
  GpLists,
  OtlSync;

procedure ProcessList(const intf: IGpIntegerList);
begin
  //...
end;

var
  lockedIntf: Locked<IGpIntegerList>;

procedure Test2;
begin
  lockedIntf := TGpIntegerList.CreateInterface;
  //...
  lockedIntf.Acquire;
  try
    ProcessList(lockedIntf);
  finally lockedIntf.Release; end;
end;
```

The interesting fact to notice is although the `lockedIntf` is declared as a variable of type `Locked<IGpIntegerList>`, it can be initialized and used as if it is of type `IGpIntegerList`. This is accomplished by providing `Implicit` operators for conversion from `Locked<T>` to `T` and back. Delphi compiler is (sadly) not smart enough to use this conversion operator in some cases so you would still sometimes have to use the provided `Value` property. For example, you'd have to do it to release wrapped object. (In the example above we have wrapped an interface and the compiler itself handled the destruction.)

```
procedure ProcessObjList(obj: TGpIntegerList);
begin
```

```
  //...
end;

var
  lockedObj: Locked<TGpIntegerList>;

procedure Test3;
begin
  lockedObj := TGpIntegerList.Create;
  try
    //...
    lockedObj.Acquire;
    try
      ProcessObjList(lockedObj);
    finally lockedObj.Release; end;
    //...
  finally lockedObj.Value.Free; end;
end;
```

Besides the standard `Acquire`/`Release` methods, `Locked<T>` also implements methods used for pessimistic locking, which is described later in this chapter, and two almost identical methods called `Locked` which allow you to execute a code segment (a procedure, method or an anonymous method) while the critical section is acquired. (In other words, you can be assured that the code passed to the `Locked` method is always executed only once provided that all code in the program properly locks access to the shared variable.)

```
type
  Locked<T> = record
  public
    type TFactory = reference to function: T;
    type TProcT = reference to procedure(const value: T);
    constructor Create(const value: T; ownsObject: boolean = true);
    class operator Implicit(const value: Locked<T>): T; inline;
    class operator Implicit(const value: T): Locked<T>; inline;
    function  Initialize(factory: TFactory): T; overload;
    {$IFDEF OTL_ERTTI}
    function  Initialize: T; overload;
    {$ENDIF OTL_ERTTI}
    procedure Acquire; inline;
    procedure Locked(proc: TProc); overload; inline;
    procedure Locked(proc: TProcT); overload; inline;
    procedure Release; inline;
    procedure Free; inline;
    property Value: T read GetValue;
  end;

procedure Locked<T>.Locked(proc: TProc);
begin
  Acquire;
  try
    proc;
  finally Release; end;
end;

procedure Locked<T>.Locked(proc: TProcT);
begin
  Acquire;
  try
    proc(Value);
  finally Release; end;
end;
```

5.1.3.1 Why not use TMonitor?

There is an alternative built into Delphi since 2009 which provides functionality similar to the `Locked<T>` – `TMonitor`. In modern Delphis, **every** object can be locked by using `System.TMonitor.Enter` function and unlocked by using `System.TMonitor.Exit`. The example above could be rewritten to use the `TMonitor` without much work.

```
var
  obj: TGpIntegerList;

procedure Test4;
begin
  obj := TGpIntegerList.Create;
  try
    //...
    System.TMonitor.Enter(obj);
    try
      ProcessObjList(obj);
    finally System.TMonitor.Exit(obj); end;
    //...
  finally FreeAndNil(obj); end;
end;
```

A reasonable question to ask is, therefore, why implementing `Locked<T>`. Why is `TMonitor` not good enough? There are plenty of reasons for that.

- `TMonitor` was buggy since its inception[8,9] (although that was fixed few years later).
- Using `TMonitor` doesn't convey your intentions. Just by looking at the variable/field declaration you wouldn't know that the entity is supposed to be used in a thread-safe manner. Using `Locked<T>`, however, explicitly declares your intent.
- `TMonitor.Enter/Exit` doesn't work with interfaces, records and primitive types. `Locked<T>` does.

On the positive size, `TMonitor` is faster than a critical section.

5.2 TOmniMREW

A typical situation in a multithreaded program is a multiple readers/exclusive writer scenario. It occurs when there are multiple reader threads which can operate on the same object simultaneously, but must be locked out when an exclusive writer thread wants to make changes to this object. Delphi already implements a synchronizer for this scenario (`TMultiReadExclusiveWriteSynchronizer`[10,11] from SysUtils), but it is quite a heavy weight object which you can use in many different ways. For situations when the probability of collision[12] is low and especially, when the object is not locked for a long period of time, a `TOmniMREW` synchronizer will give you a better performance.

```
type
  TOmniMREW = record
  public
    procedure EnterReadLock; inline;
    procedure EnterWriteLock; inline;
    procedure ExitReadLock; inline;
    procedure ExitWriteLock; inline;
  end;
```

When used in revisions up to [r1257] (which includes all OmniThreadLibrary releases up to 3.02), `TOmniMREW` must always be included in an object (as a field) because it requires that the compiler initializes it to zero before it is used. In release 1258, internal implementation was changed to enforce this initialization even when the `TOmniMREW` is used as a local variable or as a part of another record.

To use the `TOmniMREW` synchronizer, reader must call `EnterReadLock` before reading the object and `ExitReadLock` when it doesn't need the object anymore. Similarly, writer must call `EnterWriteLock` and `ExitWriteLock`.

I'd like to stress again the importance of not locking an object for long time when using `TOmniMREW`. Both `Enter` methods wait in a tight loop while waiting to get access, which can

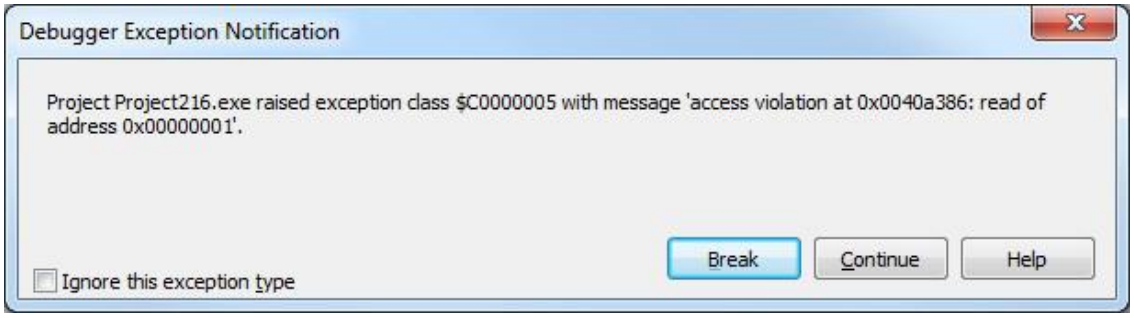
quickly use lots of CPU time if probability of collisions are high. (Collisions typically occur more often if an object is locked for extensive periods of time.)

```
procedure TOmniMREW.EnterReadLock;
var
  currentReference: NativeInt;
begin
  //Wait on writer to reset write flag so Reference.Bit0 must be 0
  //then increase Reference
  repeat
    currentReference := NativeInt(omrewReference) AND NOT 1;
  until CAS(currentReference, currentReference + 2,
            NativeInt(omrewReference));
end;

procedure TOmniMREW.EnterWriteLock;
var
  currentReference: NativeInt;
begin
  //Wait on writer to reset write flag so omrewReference.Bit0 must be 0
  //then set omrewReference.Bit0
  repeat
    currentReference := NativeInt(omrewReference) AND NOT 1;
  until CAS(currentReference, currentReference + 1,
            NativeInt(omrewReference));
  //Now wait on all readers
  repeat
  until NativeInt(omrewReference) = 1;
end;
```

Due to an optimized implementation that favours speed over safety, you'll get a cryptic access violation error if the `TOmniMREW` instance is destroyed while a read or write lock is taken. To be clear, this is a programming error; you should never destroy a synchronization object while it holds a lock. It's just that the error displayed will not make it very clear what you are doing wrong.

For example, the following test code fragment will cause an access violation.

```
procedure Test5;
var
  cs: TOmniMREW;
begin
  cs.EnterWriteLock;
end;

Test5; //<-- accvio here!
```



5.3 Cancellation token

Sometimes you want to instruct background tasks to stop whatever they are doing and quit. Typically, this happens when the program is shutting down. Programs using the "standard" multithreaded programming (i.e. `TThread`) are solving this problem each in its own way, typically by using boolean flags or Windows events.

To make the task cancellation simpler and more standardized, OmniThreadLibrary introduces a cancellation token. A cancellation token is an instance of the `IOmniCancellationToken` interface and implements functionality very similar to the Windows event synchronization primitive.

```
type
  IOmniCancellationToken = interface
    procedure Clear;
    function  IsSignalled: boolean;
    procedure Signal;
    property Handle: THandle read GetHandle;
  end; { IOmniCancellationToken }

function CreateOmniCancellationToken: IOmniCancellationToken;
```

By default, a cancellation token is in a cleared (inactive) state. To signal it, a code calls the `Signal` method. Signalled token can be cleared by calling the `Clear` method.

The task can check the cancellation token's state by calling the `IsSignalled` method or by waiting (using `WaitForSingleObject` or any of its variants) on the `Handle` property. Wait will succeed when the cancellation token is signalled.

An important part of the cancellation token implementation is that the same token can be shared between multiple tasks. To cancel all tasks, the code must only call `Signal` once (provided that other parts of the program don't call `Clear`, of course).

Cancellation tokens are used in low-level and high-level multithreading. Low-level multithreading uses the `CancelWith` method to pass a multithreading token around while the high-level multithreading uses the task configuration block.

Cancellation is demonstrated in examples `35_ParallelFor` and `38_OrderedFor`.

5.4 Waitable value

The communication framework in the OmniThreadLibrary works asynchronously (you cannot know when a task or owner will receive and process the message). Most of the time that functions great, but sometimes you have to process messages synchronously (that is, you want to wait until the task processes the message) because otherwise the code gets too complicated. For those situations, OmniThreadLibrary offers a waitable value `TOmniWaitableValue`, which is also exposed as an interface `IOmniWaitableValue`.

```
type
  IOmniWaitableValue = interface
    procedure Reset;
    procedure Signal; overload;
    procedure Signal(const data: TOmniValue); overload;
    function  WaitFor(maxWait_ms: cardinal = INFINITE): boolean;
    property Handle: THandle read GetHandle;
    property Value: TOmniValue read GetValue;
  end;

  TOmniWaitableValue = class(TInterfacedObject, IOmniWaitableValue)
  public
    constructor Create;
    destructor  Destroy; override;
    procedure Reset; inline;
    procedure Signal; overload; inline;
    procedure Signal(const data: TOmniValue); overload; inline;
    function  WaitFor(maxWait_ms: cardinal = INFINITE): boolean; inline;
    property Handle: THandle read GetHandle;
    property Value: TOmniValue read GetValue;
  end;
```

```
22
23 function  CreateWaitableValue: IOmniWaitableValue;
```

The usage pattern is quite simple. The caller creates an object or interface of that type, sends it to another thread (typically via `Task.Comm.Send`) and calls the `WaitFor` method. The task receives the message, does the processing and calls `Signal` to signal completion or `Signal(some_data)` to signal completion and return some data. At that point, the `WaitFor` returns and caller can read the data from the `Value` property.

A practical example should clarify this explanation. The two methods below are taken from the OtlThreadPool unit.

When a code wants to cancel threadpooled task, it will call the `Cancel` function. This function sends the `Cancel` message to the worker task and passes along the ID of the task to be cancelled and a `TOmniWaitableValue` object. Then it waits for the object to become signalled.

```
1  function TOmniThreadPool.Cancel(taskID: int64): boolean;
2  var
3    res: TOmniWaitableValue;
4  begin
5    res := TOmniWaitableValue.Create;
6    try
7      otpWorkerTask.Invoke(@TOTPWorker.Cancel, [taskID, res]);
8      res.WaitFor(INFINITE);
9      Result := res.Value;
10   finally FreeAndNil(res); end;
11 end;
```

The `Cancel` method in the worker task processes the message, does lots of complicated work (removed for clarity) and at the end calls the `Signal` method on the `TOmniWaitableValue` object to signal completion and return a boolean value.

```
1 procedure TOTPWorker.Cancel(const params: TOmniValue);
2 var
3   waitParam: TOmniValue;
4   wasTerminated: boolean;
5 begin
6   //... lots of code; sets wasTerminated to some value
7   waitParam := params[1];
8   (waitParam.AsObject as TOmniWaitableValue).Signal(wasTerminated);
9 end; { TOTPWorker.Cancel }
```

Very soon after the `Signal` is called, the `WaitFor` in the caller code exits and `TOmniThreadPool.Cancel` retrieves result from the `Value` property.

5.5 Inverse semaphore

A semaphore is a counting synchronisation object that starts at some value (typically greater than 0) which usually represents a number of available resources of some kind. To allocate a semaphore, one waits on it. If the semaphore count is greater than zero, the semaphore is signalled, wait will succeed and semaphore count gets decremented by one. [Of course, all of this occurs atomically.] If the semaphore count is zero, the semaphore is not signalled and wait will block until the timeout or until some other thread releases the semaphore, which increments the semaphore's count and puts it into the signalled state.

Semaphores are very useful in multithreaded programming but sadly out of scope of this book. If you are interested in their practical usage, I'd heartily recommend reading The Little Book of Semaphores, a free textbook on all things semaphorical. Multithreading – The Delphi Way also contains a chapter on semaphores.

While semaphores are implemented in the Windows kernel and Delphi wrapps them in a

pretty object TSemaphore, Windows doesn't support an useful variation on the theme – an inverse semaphore, also known as a countdown event.

Delphi implementation for this type of synchronisation primitive, TCountdownEvent, was added in release XE2.

Inverse semaphore differs from a normal semaphore by getting signalled when the count drops to zero. This allows another thread to execute a blocking wait that will succeed only when the semaphore's count is zero. Why is that good, you'll ask? Because it simplifies resource exhaustion detection. If you wait on an inverse semaphore and this semaphore becomes signalled, then you know that the resource is fully used.

The inverse semaphore is implemented by the TOmniResourceCount class which implements an IOmniResourceCount interface.

```
type
  IOmniResourceCount = interface
    function  Allocate: cardinal;
    function  Release: cardinal;
    function  TryAllocate(var resourceCount: cardinal;
      timeout_ms: cardinal = 0): boolean;
    property Handle: THandle read GetHandle;
  end;

  TOmniResourceCount = class(TInterfacedObject, IOmniResourceCount)
  public
    constructor Create(initialCount: cardinal);
    destructor  Destroy; override;
    function  Allocate: cardinal; inline;
    function  Release: cardinal;
    function  TryAllocate(var resourceCount: cardinal;
      timeout_ms: cardinal = 0): boolean;
    property Handle: THandle read GetHandle;
  end;

function CreateResourceCount(initialCount: integer): IOmniResourceCount;
```

Initial resource count is passed to the constructor or to the CreateResourceCount function. Allocate will block if this count is zero (and will unblock automatically when the count becomes greater than zero); otherwise it will decrement the count. The new value of the resource count is returned as a function result.

You should keep in mind that this number may no longer be correct when it is processed in the calling code if other threads are using the same inverse semaphore at that time.

The TryAllocate is a safer version of Allocate taking a timeout parameter (which may be set to INFINITE) and returning success/fail status as a function result.

Release increments the count and unblocks waiting Allocates. New resource count (potentially already incorrect at the moment caller sees it) is returned as the result.

Finally, there is a Handle property exposing a handle which is signalled when resource count is zero and unsignalled otherwise.

5.6 Initialization

Initializing an object in a multithreaded world is not a problem – if the object is initialized before it is shared. To put this into a simple language – everything is fine if we can initialize object first and then pass it to multiple tasks.[13]

In most cases, this is not a problem, but sometimes we want to use a shared global object in multiple tasks. In that case, the first task that wants to use the object will have to create it. While this may look as a weird approach to programming, it is a legitimate programming pattern, called lazy initialization.

The reason behind this weirdness is that sometimes we don't know in advance whether an object (or some part of a composite object) will be used at all. If the probability that the object will be used is low enough, it may be a good idea not to initialize it in advance, as that would take some time and use some memory (or maybe even lots of memory).

Additionally, there may not be a good place to call the initialization. A good example is the TOmniCS record where we want to do an implicit initialization the first time an `Acquire` method is called. As this record is usually just declared as a variable/field and not explicitly initialized, there is no better place to call then initialization code than from the `Acquire` itself.

This part of the book will explain two well-known approaches to shared initialization – a pessimistic initialization and an optimistic initialization. There's also a third approach – busy-wait – which you can read more about on my blog.

The difference between the two approaches is visible from the following pseudocode.

```
1  var
2    Shared: T;
3
4  procedure OptimisticInitializer;
5  var
6    temp: T;
7  begin
8    if not assigned(Shared) then begin
9      temp := T.Create;
10     if not AtomicallyTestAndStore(Shared, nil, temp) then
11       temp.Free;
12   end;
13 end;
14
15 procedure PessimisticInitializer;
16 begin
17   if not assigned(Shared) then begin
18     Lock;
19     try
20       if not assigned(Shared) then
21         Shared := T.Create;
22     finally Unlock; end;
23   end;
24 end;
```

An optimistic initializer assumes that there's hardly a chance of initialization being called from two tasks at the same time. Under this assumption, it is fastest to initialize the object (in the code above, the initialization is represented by creation of the shared object) and then atomically copying this object into the shared field/variable. The (nonexisting) `AtomicallyTestAndStore` method compares old value of `Shared` with `nil` and stores `temp` into `Shared` if `Shared` is nil. It makes all this in a way that prevents the code from being executed from two threads at the same time. If the `AtomicallyTestAndStore` fails (returns False), another task has already modified the `Shared` variable and we must destroy the temporary resource.

The advantage of this approach is that there is no locking so we don't have to create an additional critical section. Only CPU-level bus locking is used to implement the

`AtomicallyTestAndStore`. The disadvantage is that duplicate objects may be created at some point.

A pessimistic initializer assumes that there's a significant probability of initialization being called from two tasks at the same time and uses an additional critical section to lock access to the initialization part. A test, lock, retest pattern is used for performance reason – the code first checks whether the shared object is initialized then (if it is not) locks the critical section and retests the shared object as another task could have initialized it in the meantime.

The advantage of this approach is that only a single object is created. The disadvantage is that we must manage additional critical section that will be used for locking.

It is unclear which approach is better. Although locking slows the application more than microlocking, creating duplicate resources may slow it down even more. On the other hand, pessimistic initializer requires additional lock, but that won't make much difference if you don't create millions of shared objects. In most cases initialization code will be rarely called and the complexity of initializer will not change the program performance in any meaningful way so the choice of initializer will mainly be a matter of personal taste.

5.6.1 Pessimistic initialization

While pessimistic initialization doesn't represent any problems for a skilled programmer, it is bothersome as we must manage an additional locking object. (Typically that will be a critical section.) To simplify the code and to make it more intentional, OmniThreadLibrary introduces a `Locked<T>` type which wraps any type (the type of your shared object) and a critical section.

An instance of the `Locked<T>` type contains two fields – one holding your data (`FValue`) and another containing a critical section (`FLock`). `Locked<T>` provides two helper functions (`Initialize`) which implement the pessimistic initialization pattern.

The first version accepts a factory function which creates the object. The code implements the test, lock, retest pattern explained previously in this section.

```
1  function Locked<T>.Initialize(factory: TFactory): T;
2  begin
3    if not FInitialized then begin
4      Acquire;
5      try
6        if not FInitialized then begin
7          FValue := factory();
8          FInitialized := true;
9        end;
10     finally Release; end;
11   end;
12   Result := FValue;
13 end;
```

Another version, implemented only in Delphi 2010 and newer, doesn't require a factory function but calls the default (parameter-less) constructor. This is, of course, only possible if the `T` type represents a class. Actually, this method simply calls the other version and provides a special factory method which travels the extended RTTI information, selects an appropriate constructor and executes it to create the shared object.

```
function Locked<T>.Initialize: T;
begin
  if not FInitialized then begin
    if PTypeInfo(TypeInfo(T))^.Kind  <> tkClass then
      raise Exception.Create('Locked<T>.Initialize: Unsupported type');
    Result := Initialize(
      function: T
      var
        aMethCreate : TRttiMethod;
        instanceType: TRttiInstanceType;
        ctx         : TRttiContext;
        params      : TArray<TRttiParameter>;
        resValue    : TValue;
        rType       : TRttiType;
      begin
        ctx := TRttiContext.Create;
        rType := ctx.GetType(TypeInfo(T));
        for aMethCreate in rType.GetMethods do begin
          if aMethCreate.IsConstructor then begin
            params := aMethCreate.GetParameters;
            if Length(params) = 0 then begin
              instanceType := rType.AsInstance;
              resValue := AMethCreate.Invoke(
                            instanceType.MetaclassType, []);
              Result := resValue.AsType<T>;
              break; //for
            end;
          end;
        end; //for
      end);
  end;
end;
```

`Locked<T>` also implements methods `Acquire` and `Release` which use the built-in critical section to implement synchronization.

5.6.2 Optimistic initialization

An optimistic initialization is supported with the `Atomic<T>` class which is much simpler than the pessimistic `Locked<T>` alternative.

```
type
  Atomic<T> = class
    type TFactory = reference to function: T;
    class function Initialize(var storage: T;
      factory: TFactory): T; overload;
    {$IFDEF OTL_ERTTI}
    class function Initialize(var storage: T): T; overload;
    {$ENDIF OTL_ERTTI}
  end;
```

As in `Locked<T>`, there are two `Initialize` functions, one creating the object using a user-provided factory function and another using RTTI to call the default parameter-less constructor. We'll only examine the former.

The second overload is only available in Delphi 2010 and newer.

```
class function Atomic<T>.Initialize(var storage: T; factory: TFactory): T;
var
  interlockRes: pointer;
  tmpT        : T;
begin
  if not assigned(PPointer(@storage)^) then begin
    Assert(cardinal(@storage) mod SizeOf(pointer) = 0,
      'Atomic<T>.Initialize: storage is not properly aligned!');
```

```
    Assert(cardinal(@tmpT) mod SizeOf(pointer) = 0,
      'Atomic<T>.Initialize: tmpT is not properly aligned!');
    tmpT := factory();
    interlockRes := InterlockedCompareExchangePointer(
                      PPointer(@storage)^, PPointer(@tmpT)^, nil);
    case PTypeInfo(TypeInfo(T))^.Kind of
      tkInterface:
        if interlockRes = nil then
          PPointer(@tmpT)^ := nil;
      tkClass:
        if interlockRes <> nil then
          TObject(PPointer(@tmpT)^).Free;
      else
        raise Exception.Create('Atomic<T>.Initialize: Unsupported type');
    end; //case
  end;
  Result := storage;
end;
```

The code first checks if the storage is already initialized by using a weird cast which assumes that the `T` is pointer-sized. This is a safe assumption because `Atomic<T>` only supports `T` being a class or an interface.

Next the code checks whether the shared object and the temporary variable are properly aligned. This should in most cases not present a problem as all 'normal' fields (not stored in `packed record` types) should always be appropriately aligned.

After that, the factory function is called to create an object.

Next, the `InterlockedCompareExchangePointer` is called. It takes three parameters – a destination address, an exchange data and a comparand. The functionality of the code can be represented by the following pseudocode:

```
function InterlockedCompareExchangePointer(var destination: pointer;
  exchange, comparand: pointer): pointer;
begin
  Result := destination;
  if destination = comparand then
    destination := exchange;
end;
```

The trick here is that this code is all executed inside the CPU, atomically. The CPU ensures that the destination value is not modified (by another CPU) during the execution of the code. It is hard to understand (interlocked functions always make my mind twirl in circles) but basically it reduces to two scenarios:

- Function returns `nil` (old, uninitialized value of `storage`), and `storage` is set to new object (`tmpT`).
- Function returns already existing object (old, just initialized value of `storage`) and `storage` is not modified.

In yet another words – `InterlockedCompareExchangePointer` either stores new value in the `storage` and returns `nil` or does nothing, leaves already initialized `storage` intact and returns something else than `nil`.

At the end, the code handles two specific case. If a `T` is an interface type and initialization was successful, the temporary value in `tmpT` must be replaced with `nil`. Otherwise two variables (`storage` and `tmpT`) would own an interface with a reference count of 1 which would cause big problems.

```
if interlockRes = nil then
  PPointer(@tmpT)^ := nil;
```

It a `T` is a class type and initialization was **not** successful, the temporary value stored in `tmpT` must be destroyed.

```
if interlockRes <> nil then
  TObject(PPointer(@tmpT)^).Free;
```

`Initialize` returns the same shared object twice – once in the `storage` parameter and once as a result of the function. This allows us to write very space-efficient initializers like in the example below, taken from the OtlParallel unit.

```
function TOmniWorkItem.GetCancellationToken: IOmniCancellationToken;
begin
  Result := Atomic<IOmniCancellationToken>.Initialize(
    FCancellationToken, CreateOmniCancellationToken);
end;
```

When you are initializing an interface and a new instance of the implementing object is created by calling the default constructor `Create`, you can use the two-parameter version of `Atomic` to simplify the code. [3.06] This is only supported in Delphi XE and newer.

```
  Atomic<I; T:constructor> = class
    class function Initialize(var storage: I): I;
  end;
```

For example, if the shared object is stored in `shared: IMyInterface` and is created by calling `TMyInterface.Create`, you can initialize it via:

```
Atomic<IMyInterface, TMyInterface>.Initialize(shared);
```

5.7 TWaitFor

A common scenario in parallel programming is that the program has to wait for something to happen. The occurrence of that something is usually signalled with an event.

On Windows, this is usually accomplished by calling one of the functions from the `WaitForMultipleObjects` family. While they are pretty powerful and quite simple to use, they also have a big limitation – one can only wait for up to 64 events at the same time.

Windows also offers a `RegisterWaitForSingleObject` API call which can be used to circumvent this limitation. Its use is, however, quite complicated to use. To simplify programmer's life, OmniThreadLibrary introduces a `TWaitFor` class which allows the code to wait for any number of events.

```
type
  TWaitFor = class
  public type
    TWaitResult = (
      waAwaited,      // WAIT_OBJECT_0 .. WAIT_OBJECT_n
      waTimeout,      // WAIT_TIMEOUT
      waFailed,       // WAIT_FAILED
      waIOCompletion // WAIT_IO_COMPLETION
    );
    THandleInfo = record
      Index: integer;
    end;
    THandles = array of THandleInfo;

```

```
15    constructor Create; overload;
16    constructor Create(const handles: array of THandle); overload;
17    destructor  Destroy; override;
18    function  MsgWaitAny(timeout_ms, wakeMask, flags: cardinal): TWaitResult;
19    procedure SetHandles(const handles: array of THandle);
20    function  WaitAll(timeout_ms: cardinal): TWaitResult;
21    function  WaitAny(timeout_ms: cardinal; alertable: boolean = false): TWaitResult;
22    property Signalled: THandles read FSignalledHandles;
23  end;
```

To use `TWaitFor`, you have to create an instance of this class and pass it an array of handles either as a constructor parameter or by calling the `SetHandles` method. All handles must be created with the `CreateEvent` Windows function.

You can then wait for any (`WaitAny`) or all (`WaitAll`) events to become signalled. In both cases the `Signalled` array is filled with information about signalled (set) events. The `Signalled` property is an array of `THandleInfo` records, each of which (currently) only contains one field - an index (into the `handles` array) of the signalled event.

For example, if you want to wait for two events and then react to them, you should use the following approach:

```
1 var
2   wf: TWaitFor;
3   info: THandleInfo;
4
5 wf := TWaitFor.Create([handle1, handle2]);
6 try
7   if wf.WaitAny(INFINITE) = waAwaited then begin
8     for info in wf.Signalled do
9       if info.Index = 0 then
10        // handle1 is signalled - do something
11      else if info.Index = 1 then
12        // handle2 is signalled - do something
13  end;
14 finally FreeAndNil(wf); end;
```

You don't have to recreate `TWaitFor` for each wait operation; it is perfectly ok to call `WaitXXX` functions repeatedly on the same object. It is also fine to change the array of handles between two `WaitXXX` calls by calling the `SetHandles` method.

The `WaitAny` method also comes in a variant which processes Windows messages, I/O completion routines and APC calls (`MsgWaitAny`). Its `wakeMask` and `flags` parameters are the same as the corresponding parameters to the `MsgWaitForMultipleObjectsEx` API.

The use of the `TWaitFor` is shown in demo `59_TWaitFor`.

5.8 TOmniLockManager<K>

The `TOmniLockManager<K>` class solves a very specific problem – how to synchronize access to entities of any type. In a way, it is similar to `TMonitor`, except that it works on all types, not just on objects.

If you need to lock access to objects, `TMonitor` should be used instead as it is much faster than the `TOmniLockManager<K>`.

Following requirements are implemented in the `TOmniLockManager<K>`.

1. The lock manager (LM for short) is used to lock entities of any type, not just objects. Let's name such an entity a key.
2. LM works similarly to a mutex. The caller can wait on a key with a timeout. Timeouts of 0 and INFINITE are supported.
3. The interface is similar to `TMonitor.Enter/Exit`. The code calls `Lock` to get exclusive access to a key and calls `Unlock` to release the key back to the public use.
4. The LM is reentrant. If a thread manages to get a lock on a key, it will always succeed in getting another lock before releasing the first one. Only when the number of `Unlock` calls in one thread matches the number of `Lock` calls, the key is unlocked. (In other words, if you call `Lock` twice with the same key, you also have to call `Unlock` twice to release that key.)
5. An exclusive access is granted fairly. If a thread A has started waiting on a key before the thread B, it will get access to that key before the thread B.
6. The set of keys is potentially huge so it is not possible to create a critical section/mutex for each key.
7. A probability of collision (two threads trying to lock the same resource at the same time) is fairly low.

A series of three articles on my blog (ThreadSafe Lock Manager: Design, Code, Test) describes the design issues and code implementation in more details.

```
type
  IOmniLockManagerAutoUnlock = interface
    procedure Unlock;
  end;

  IOmniLockManager<K> = interface
    function  Lock(const key: K; timeout_ms: cardinal): boolean;
    function  LockUnlock(const key: K; timeout_ms: cardinal): IOmniLockManagerAutoUnlock;
    procedure Unlock(const key: K);
  end;
```

The `TOmniLockManager<K>` public class implements the `IOmniLockManager<K>` interface.

The `Lock` function returns `False` if it fails to lock the key in the specified timeout. Timeouts 0 and `INFINITE` are supported.

There's also a `LockUnlock` function which returns an interface that automatically unlocks the key when it is released. This interface also implements an `Unlock` function which unlocks the key.

A practical example of using the lock manager is shown in demo `54_LockManager`.

5.9 TOmniSingleThreadUseChecker

For debugging purpuses, OmniThreadLibrary implements the `TOmniSingleThreadUseChecker` record. It gives the programmer a simple way to make sure that some code is always executed from the same thread.

```
type
  TOmniSingleThreadUseChecker = record
    procedure AttachToCurrentThread;
    procedure Check;
    procedure DebugCheck;
  end;
```

Using it is simple – first declare a variable/field of type `TOmniSingleThreadUseChecker` in a context that has to be checked and then call `Check` or `DebugCheck` method of that variable whenever you want to check that some part of code was not used from more than one thread.

The difference between `Check` and `DebugCheck` is that the latter can be disabled during the compilation. It implements the check only if the conditional symbol `OTL_CheckThreadSafety` is defined. Otherwise, `DebugCheck` contains no code and does not affect the execution speed.

In cases where you do use such an object from more than one thread (for example, if you use it from a task and then from the task controller after the task terminates) you can call `AttachToCurrentThread` to associate the checker with the current thread.

A. Units

6. Miscellaneous

This chapter covers some OmniThreadLibrary classes, records, and interfaces, that are useful for everyday programming but somehow did not find place in any other chapter.

6.1 TOmniTwoWayChannel

The OtlComm unit implements a `TOmniTwoWayChannel` class with a corresponding `IOmniTwoWayChannel` interface, which defines a bidirectional communication channel. This channel consists of two unidirectional channels, which are exposed through two `IOmniCommunicationEndpoint` interfaces.

```
type
  IOmniTwoWayChannel = interface ['{3ED1AB88-4209-4E01-AA79-A577AD719520}']
    function Endpoint1: IOmniCommunicationEndpoint;
    function Endpoint2: IOmniCommunicationEndpoint;
  end;

  TOmniTwoWayChannel = class(TInterfacedObject, IOmniTwoWayChannel)
    constructor Create(messageQueueSize: integer; taskTerminatedEvent: THandle);
    destructor  Destroy; override;
    function Endpoint1: IOmniCommunicationEndpoint; inline;
    function Endpoint2: IOmniCommunicationEndpoint; inline;
  end;

  function CreateTwoWayChannel(numElements: integer = CDefaultQueueSize;
    taskTerminatedEvent: THandle = 0): IOmniTwoWayChannel;
```

This interface is used internally for task controller/task communication, but can be also used in your own code.

One side on communication should use `Endpoint1` endpoint and its `Send`/ `Receive` methods while the other side should use the `Endpoint2` endpoint and its `Send`/`Receive` methods. Whatever is sent to `Endpoint1` can be received on `Endpoint2` and whatever is sent to `Endpoin2` can be received on `Endpoint1`.

See also: `RegisterComm`/`UnregisterComm` in sections `RegisterComm` and `IOmniTask` interface. Another example can be found in the `8_RegisterComm` demo.

6.2 TOmniValueContainer

The `TOmniValueContainer` class implements a growable list of `TOmniValue` values, indexed with an integer and string values. It is used internally in the `TOmniValue.AsArray` and for task parameter passing.

```
type
  TOmniValueContainer = class
  public
    constructor Create;
    procedure Add(const paramValue: TOmniValue; paramName: string = '');
    procedure Assign(const parameters: array of TOmniValue);
    procedure AssignNamed(const parameters: array of TOmniValue);
    function  ByName(const paramName: string): TOmniValue; overload;
    function  ByName(const paramName: string;
      const defValue: TOmniValue): TOmniValue; overload;
    function  Count: integer;
    function  Exists(const paramName: string): boolean;
    function  IndexOf(const paramName: string): integer;
    procedure Insert(paramIdx: integer; const value: TOmniValue);
    function  IsLocked: boolean; inline;
```

```
    procedure Lock; inline;
    property Item[paramIdx: integer]: TOmniValue read GetItem write SetItem; default;
    property Item[const paramName: string]: TOmniValue read GetItem write SetItem; default;
    property Item[const param: TOmniValue]: TOmniValue read GetItem write SetItem; default;
  end;
```

`Add` adds a new value to the list. Index can be an integer value (starting with 0) or a string value. This method will raise an exception if the container is locked (see below).

`Assign` assigns an array of `TOmniValue`s to the container. Previous values stored in the container are lost (see the example for `TOmniValue.CreateNamed` for more information). This method will raise an exception if the container is locked (see below).

`AssignNamed` assigns an array of named values to the container. Elements of the array must alternate between names (string indexes) and values. Previous values stored in the container are lost. This method will raise an exception if the container is locked (see below).

`ByName` searches for a value with the specified name and returns the value. Searching is linear. Because of that, `TOmniValueContainer` should not be used to store large quantity of string-indexed data. One version of the function returns `TOmniValue.Null` if the string key is not found, while the other returns the default value, passed to the function call.

`Count` returns the current size of the container.

`Exists` checks whether a string-indexed value with the specified name is stored in the container.

`IndexOf` returns an integer index associated with the string-indexed value. This index can be used to access the value in the container. If the value is not found, the function returns -1.

`Insert` inserts new value into an integer-indexed array. This method will raise an exception if the container is locked (see below).

`IsLocked` checks where the container is locked. Locked container will not accept new values.

`Lock` locks the container. That prevents the code from changing the container. From that point, values can only be read from the container, not written to. A container cannot be unlocked.

`Item` property accesses a specific value either by an integer index (from 0 to `Count-1`), or by string index. If a `TOmniValue` is passed as an index, the type of the `TOmniValue` index parameter will determine how the container is accessed.

6.3 TOmniCounter

The `CreateCounter` (OtlCommon unit) function creates a counter with an atomic increment and decrement operations. Such counter can be used from multiple threads at the same time without any locking. Accessing the counter's value is also thread-safe.

The counter is returned as an `IOmniCounter` interface. It is implemented by the `TOmniCounter` class, which you can use in your code directly if you'd rather deal with objects than interfaces.

```
type
  IOmniCounter = interface
    function  Increment: integer;
    function  Decrement: integer;
    function  Take(count: integer): integer; overload;
    function  Take(count: integer; var taken: integer): boolean; overload;
    property Value: integer read GetValue write SetValue;
```

```
8   end;
9
10  TOmniCounter = record
11    procedure Initialize;
12    function  Increment: integer;
13    function  Decrement: integer;
14    function  Take(count: integer): integer; overload;
15    function  Take(count: integer; var taken: integer): boolean; overload;
16    property Value: integer read GetValue write SetValue;
17  end;
18
19  function  CreateCounter(initialValue: integer = 0): IOmniCounter;
```

The counter part of the `TOmniCounter` record is automatically initialized on the first use. If you want, you can call `Initialize` in advance, although that is not required.

`Take` is a special operation which tries to decrement the counter by `count` but stops at 0. It returns the number that could be taken from the counter (basically, `Min(count, counter.Value)`). Its effect is the same as the following code (except that the real implementation of `Take` is thread-safe).

```
1 Result := Min(counter, count);
2 counter := counter - Result;
```

`Take` is used in demo Parallel Data Production.

6.4 TOmniAlignedInt32 and TOmniAlignedInt64

Those two records hold 4-byte (32 bit) and 8-byte (64 bit) values, respectively. These values are suitably aligned so it can be read from and written to in an atomic operation. They also implement atomic `Increment`, `Decrement`, `Add`, and `Substract` operations.

These two types were added in version [3.06]. Previously, OmniThreadLibrary used functionally equivalent types `TGp4AlignedInt` and `TGp8AlignedInt64` from the GpStuff unit.

Reading and writing values stored in the record (through the `Value` property or by using a supplied `Implicit` operator) is also atomic on the Win64 platform.

```
1  type
2    TOmniAlignedInt32 = record
3    public
4      procedure Initialize; inline;
5      function  Add(value: integer): integer; inline;
6      function  Addr: PInteger; inline;
7      function  CAS(oldValue, newValue: integer): boolean;
8      function  Decrement: integer; overload; inline;
9      function  Decrement(value: integer): integer; overload; inline;
10     function  Increment: integer; overload; inline;
11     function  Increment(value: integer): integer; overload; inline;
12     function  Subtract(value: integer): integer; inline;
13     class operator Add(const ai: TOmniAlignedInt32; i: integer): cardinal; inline;
14     class operator Equal(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
15     class operator GreaterThan(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
16     class operator GreaterThanOrEqual(const ai: TOmniAlignedInt32; i: integer): boolean;
17       inline;
18     class operator Implicit(const ai: TOmniAlignedInt32): integer; inline;
19     class operator Implicit(const ai: TOmniAlignedInt32): cardinal; inline;
20     class operator Implicit(const ai: TOmniAlignedInt32): PInteger; inline;
21     class operator LessThan(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
22     class operator LessThanOrEqual(const ai: TOmniAlignedInt32; i: integer): boolean;
23       inline;
24     class operator NotEqual(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
25     class operator Subtract(ai: TOmniAlignedInt32; i: integer): cardinal; inline;
26     property Value: integer read GetValue write SetValue;
27   end;
```

```
28
29  TOmniAlignedInt64 = record
30  public
31    procedure Initialize; inline;
32    function  Add(value: int64): int64; inline;
33    function  Addr: PInt64; inline;
34    function  CAS(oldValue, newValue: int64): boolean;
35    function  Decrement: int64; overload; inline;
36    function  Decrement(value: int64): int64; overload; inline;
37    function  Increment: int64; overload; inline;
38    function  Increment(value: int64): int64; overload; inline;
39    function  Subtract(value: int64): int64; inline;
40    property Value: int64 read GetValue write SetValue;
41  end;
```

6.5 TOmniRecordWrapper

The `TOmniRecordWrapper<T>` class allows you to wrap any record inside an instance of a class.

```
1 type
2   TOmniRecordWrapper<T> = class
3   public
4     constructor Create(const value: T);
5     function  GetRecord: T;
6     procedure SetRecord(const value: T);
7     property Value: T read GetRecord write SetRecord;
8   end;
```

You can then use the resulting object in situations where an object is expected (for example, you can store such object in a `TObjectList`).

This class is used in `TOmniValue` to implement functions `FromRecord<T>` and `ToRecord<T>`.

6.6 TOmniRecord

The `TOmniRecord<T>` record allows you to wrap any value inside a record type.

```
1 type
2   TOmniRecord<T> = record
3   strict private
4     FValue: T;
5   public
6     constructor Create(const aValue: T);
7     property Value: T read FValue write FValue;
8   end;
```

6.7 IOmniAutoDestroyObject

You can use the `CreateAutoDestroyObject` function (OtlCommon unit) to wrap any object into an `IOmniAutoDestroyObject` interface.

When this interface' s reference count drops to 0, it automatically destroys the wrapped object.

```
1 type
2   IOmniAutoDestroyObject = interface
3     property Value: TObject read GetValue;
4   end;
5
6   function  CreateAutoDestroyObject(obj: TObject): IOmniAutoDestroyObject;
```

Original object can be accessed through the `Value` property.

6.8 IOmniIntegerSet

The `IOmniIntegerSet` interface and its implementing class `TOmniIntegerSet` [3.06] can be used to store a set of integers. They are defined in the OtlCommon unit.

The big difference between the Delphi built-in `set` type and `IOmniIntegerSet` is that the latter can store more than 256 elements and that they are not limited in size. Delphi's `set` is, on the other hand, faster and supports set operations (union, intersection, difference).

Data is stored in a very wasteful manner as a TArray<boolean>. The higher bound of this array is equal to the largest number stored in the set so `IOmniIntegerSet` shouldn't be used for storing large integers. `IOmniIntegerSet` was created to store processor affinity values in the `Environment` object and its implementation is optimized for this purpose.

`IOmniIntegerSet` cannot store negative values.

```
type
  TOmniIntegerSetChangedEvent = procedure(const intSet: IOmniIntegerSet) of object;

  IOmniIntegerSet = interface
    function  Add(value: integer): boolean;
    procedure Assign(const value: IOmniIntegerSet);
    procedure Clear;
    function  Contains(value: integer): boolean;
    function  Count: integer;
    function  IsEmpty: boolean;
    function  Remove(value: integer): boolean;
  {$IFDEF OTL_HasArrayOfT}
    property AsArray: TArray<integer> read GetAsArray write SetAsArray;
  {$ENDIF OTL_HasArrayOfT}
    property AsBits: TBits read GetAsBits write SetAsBits;
    property AsIntArray: TIntegerDynArray read GetAsIntArray write SetAsIntArray;
    property AsMask: int64 read GetAsMask write SetAsMask;
    property OnChange: TOmniIntegerSetChangedEvent read GetOnChange write SetOnChange;
    property Item[idx: integer]: integer read GetItem; default;
  end;
```

`Add` adds an element to the set and returns `True` if element was previously present in the set, `False` if not.

`Assign` assigns one set to another.

`Clear` removes all elements from the set.

`Contains` checks whether an element is present in the set.

`Count` returns number of elements in the set.

`IsEmpty` returns `True` when set contains no elements.

`Remove` removes an element from the set and returns `True` if element was previously present in the set, `False` if not.

`AsArray` is available only on Delphi 2010 an newer and allows the set to be accessed as a `TArray<integer>`.

`AsBits` allows the code to access the set as Delphi's `TBits` class.

`AsIntArray` allows the code to access the set as an `array of integer`.

`AsMask` allows the code to access the set as a bitfield mask. This is possible only if all values in the set are smaller than 64.

`Item[]` allows the code to access the set as an indexed array of values, for example:

```
for i := 0 to omniSet.Count - 1 do
  DoSomethingWith(omniSet[i]);
```

`OnChange` event is triggered each time the set is modified.

6.9 Environment

The OtlCommon unit implements function `Environment` which returns a global `IOmniEnvironment` singleton. This interface can be used to access information about the system, current process, and current thread.

```
type
  IOmniEnvironment = interface
  {$IFDEF OTL_NUMASupport}
    property NUMANodes: IOmniNUMANodes read GetNUMANodes;
    property ProcessorGroups: IOmniProcessorGroups read GetProcessorGroups;
  {$ENDIF OTL_NUMASupport}
    property Process: IOmniProcessEnvironment read GetProcess;
    property System: IOmniSystemEnvironment read GetSystem;
    property Thread: IOmniThreadEnvironment read GetThread;
  end;

function  Environment: IOmniEnvironment;
```

The `System` property allows you to get the number of processors (`Affinity`).

```
type
  IOmniSystemEnvironment = interface
    property Affinity: IOmniAffinity read GetAffinity;
  end;
```

The `Process` property gives you access to the number of processors, associated with the current process (`Affinity`), memory usage statistics (`Memory`), process priority (`PriorityClass`), and execution times (`Times`).

You can change number of cores associated with the process by using the methods of the `Affinity` interface. All other information is read-only.

```
type
  // from DSiWin32.pas
  _PROCESS_MEMORY_COUNTERS = packed record
    cb: DWORD;
    PageFaultCount: DWORD;
    PeakWorkingSetSize: DWORD;
    WorkingSetSize: DWORD;
    QuotaPeakPagedPoolUsage: DWORD;
    QuotaPagedPoolUsage: DWORD;
    QuotaPeakNonPagedPoolUsage: DWORD;
    QuotaNonPagedPoolUsage: DWORD;
    PagefileUsage: DWORD;
    PeakPagefileUsage: DWORD;
  end;
  PROCESS_MEMORY_COUNTERS = _PROCESS_MEMORY_COUNTERS;
  PPROCESS_MEMORY_COUNTERS = ^_PROCESS_MEMORY_COUNTERS;
  TProcessMemoryCounters = _PROCESS_MEMORY_COUNTERS;
  PProcessMemoryCounters = ^_PROCESS_MEMORY_COUNTERS;

  // from OtlCommon.pas
  TOmniProcessMemoryCounters = TProcessMemoryCounters;

  TOmniProcessPriorityClass = (pcIdle, pcBelowNormal, pcNormal,
    pcAboveNormal, pcHigh, pcRealtime);

  TOmniProcessTimes = record
```

```
27     CreationTime: TDateTime;
28     UserTime    : int64;
29     KernelTime : int64;
30   end;
31
32   IOmniProcessEnvironment = interface
33     property Affinity: IOmniAffinity read GetAffinity;
34     property Memory: TOmniProcessMemoryCounters read GetMemory;
35     property PriorityClass: TOmniProcessPriorityClass read GetPriorityClass;
36     property Times: TOmniProcessTimes read GetTimes;
37   end;
```

The `Thread` property gives the programmer access to the number of processors, associated with the current process (`Affinity`), and to the thread ID (`ID`). You can change number of cores associated with the process by using the methods of the `Affinity` interface.

On parallel systems with multiple processor groups[14] you can use the `GroupAffinity` property to read or set group affinity for the current thread.

```
1 type
2   IOmniThreadEnvironment = interface
3     property Affinity: IOmniAffinity read GetAffinity;
4     property GroupAffinity: TOmniGroupAffinity read GetGroupAffinity
5       write SetGroupAffinity;
6     property ID: cardinal read GetID;
7   end;
```

The `ProcessorGroups` property [3.06] is available only on Delphi 2009 and newer and gives you access to the information about processor groups in the computer.

```
1 type
2   IOmniProcessorGroups = interface
3     function  All: IOmniIntegerSet;
4     function  Count: integer;
5     function  FindGroup(groupNumber: integer): IOmniProcessorGroup;
6     function  GetEnumerator: TList<IOmniProcessorGroup>.TEnumerator;
7     property Item[idx: integer]: IOmniProcessorGroup read GetItem; default;
8   end;
```

`All` returns set of all processor group numbers.

`Count` returns number of processor groups in the system.

`FindGroup` locates a group by its number.

`GetEnumerator` allows you to use a `for..in` enumerator to access all processor groups.

`Item[]` returns information on a specific processor group (0 .. `Count - 1`).

For each processor group you can retrieve its number (`GroupNumber`) and processor affinity (`Affinity`).

```
1 type
2   IOmniProcessorGroup = interface
3     property GroupNumber: integer read GetGroupNumber;
4     property Affinity: IOmniIntegerSet read GetAffinity;
5   end;
```

The `NUMANodes` property [3.06] is available only on Delphi 2009 and newer and gives you access to the information about NUMA nodes[15] in the computer.

```
1 type
2   IOmniNUMANodes = interface
```

```
    function  All: IOmniIntegerSet;
    function  Count: integer;
    function  Distance(fromNode, toNode: integer): integer;
    function  FindNode(nodeNumber: integer): IOmniNUMANode;
    function  GetEnumerator: TList<IOmniNUMANode>.TEnumerator;
    property Item[idx: integer]: IOmniNUMANode read GetItem; default;
  end;
```

`All` returns set of all NUMA node numbers in the system.

`Count` returns number of NUMA nodes in the system.

`Distance` returns relative distance between nodes.[16]

`FindNode` locates a node by its number.

`GetEnumerator` allows you to use a `for..in` enumerator to access all NUMA nodes.

`Item[]` returns information on a specific NUMA node (0 .. `Count - 1`).

For each NUMA node you can retrieve its number (`NodeNumber`), the number of processor group this NUMA node belongs to (`GroupNumber`) and processor affinity (`Affinity`).

```
  IOmniNUMANode = interface
    property NodeNumber: integer read GetNodeNumber;
    property GroupNumber: integer read GetGroupNumber;
    property Affinity: IOmniIntegerSet read GetAffinity;
  end;
```

6.9.1 IOmniAffinity

The `IOmniAffinity` interface gives you a few different ways of modifying the number of processing cores, associated with the process or thread. It also allows you to read the information about processing cores in the system.

```
type
  IOmniAffinity = interface
    property AsString: string read GetAsString write SetAsString;
    property Count: integer read GetCount write SetCount;
    property CountPhysical: integer read GetCountPhysical;
    property Mask: DSiNativeUInt read GetMask write SetMask;
  end;
```

The `AsString` property returns active (associated) cores as a string of characters. Following ansi characters are used for cores from 0 to 63 (maximum number of cores available to a Win64 application).

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz@$
```

You can change associated cores by assigning to this property.

Example: The following program will force the current program to run only on cores 2, 4, 5, and 17 (provided that they exist in the system, of course).

```
Environment.Process.Affinity.AsString := '245H';
```

The `Count` property returns number of active (associated) cores. You can also assign a number to this property to change the number of associated cores. If you do that, the code uses a random number generator to select associated cores.

The `CountPhysical` property returns number of physical (non hyper-threaded) cores,

associated with the current entity (system, process, or thread). This information is only available on Windows XP SP3 or newer. The value returned for `Count` will be used on older systems. On all other platforms it will return the same value as the `Count` property.

The `Mask` property returns a bitmask of active (associated) cores. Bit 0 represents the CPU 0 and so on. You can also change associated cores by assigning to this property.

A. Units

7. How-to

This part of the book contains practical examples of OmniThreadLibrary usage. Each of them starts with a question that introduces the problem and continues with the discussion of the solution.

Following topics are covered:

- Background file scanning

Scanning folders and files in a background thread.

- Web download and database storage

Multiple workers downloading data and storing it in a single database.

- Parallel for with synchronized output

Redirecting output from a parallel for loop into a structure that doesn't support multithreaded access.

- Using taskIndex and task initializer in parallel for

Using `taskIndex` property and task initializer delegate to provide a per-task data storage in Parallel for.

- Background worker and list partitioning

Writing server-like background processing.

- Parallel data production

Multiple workers generating data and writing it into a single file.

- Building a connection pool

Using OmniThreadLibrary to create a pool of database connections.

- QuickSort and parallel max

How to sort an array and how to process an array using multiple threads.

- Parallel search in a tree

Finding data in a tree.

- Multiple workers with multiple frames

Graphical user interface containing multiple frames where each frame is working as a frontend for a background task.

- OmniThreadLibrary and databases

Using databases from OmniThreadLibrary.

- OmniThreadLibrary and COM/OLE

Using COM/OLE from OmniThreadLibrary.

- Using message queue with a TThread worker

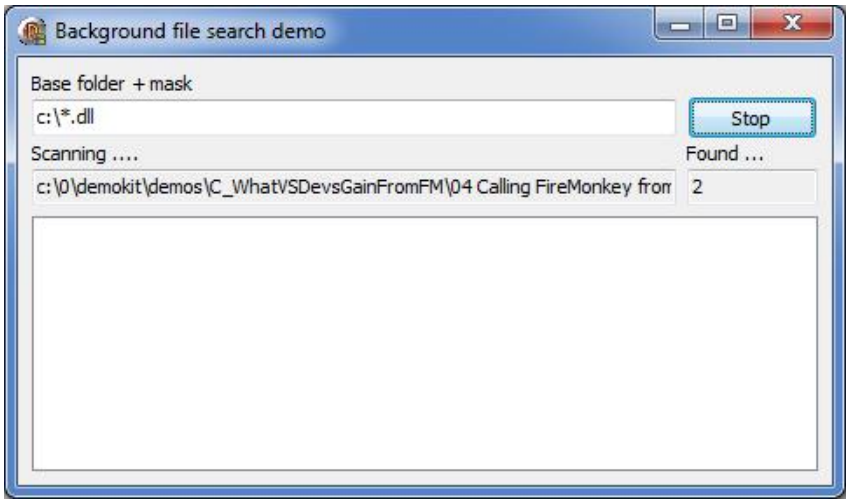
Using OmniThreadLibrary's `TOmniMessageQueue` to communicate with a TThread worker.

7.1 Background file scanning

I'm looking into a way to do a longer operation (e.g. searching files on a drive) inside a thread to keep the main application responsive. With Delphi's `TThread` I would override the `Execute` method. But that's not good enough as the normal user would like to see a progress bar. It would also be great if the user could abort the search.

This solution uses low-level part of the OmniThreadLibrary to implement a file scanner application. It is also available as a demo application `23_BackgroundFilesearch`.

The user interface is intentionally simple.



User enters a path and file mask in the edit field and clicks Scan. The application starts a background task which scans the file system and reports found files back to the application where they are displayed in the listbox.

During the scanning, main thread stays responsive. You can move the program around, resize it, minimize, maximize and so on.

Besides the components that are visible at runtime, the form contains two components – a `TOmniEventMonitor` named `OTLMonitor` and a `TTimer` called `tmrDisplayStatus`.

When the user clicks the Scan button, a background task is created.

```
1  procedure TfrmBackgroundFileSearchDemo.btnScanClick(Sender: TObject);
2  begin
3    FFileList := TStringList.Create;
4    btnScan.Enabled := false;
5    tmrDisplayStatus.Enabled := true;
6    FScanTask := CreateTask(ScanFolders, 'ScanFolders')
7      .MonitorWith(OTLMonitor)
8      .SetParameter('FolderMask', inpFolderMask.Text)
9      .Run;
10 end;
```

`ScanFolders` is the method that will do the scanning (in a background thread). Task will be monitored with the `OTLMonitor` component which will receive task messages. `OTLMonitor` will also tell us when the task has terminated. Input folder and mask are sent to the task as a

parameter `FolderMask` and task is started.

The `FFileList` field is a `TStringList` that will contain a list of all found files.

Let's ignore the scanner details for the moment and skip to the end of the scanning process. When the task has completed its job, `OTLMonitor.OnTaskTerminated` is called.

```
procedure TfrmBackgroundFileSearchDemo.OTLMonitorTaskTerminated(
  const task: IOmniTaskControl);
begin
  tmrDisplayStatus.Enabled := false;
  outScanning.Text := '';
  outFiles.Text := IntToStr(FFileList.Count);
  lbFiles.Clear;
  lbFiles.Items.AddStrings(FFileList);
  FreeAndNil(FFileList);
  FScanTask := nil;
  btnScan.Enabled := true;
end;
```

At that point, number of found files is copied to the outFiles edit field and complete list is assigned to the listbox. Task reference `FScanTask` is then cleared, which causes the task object to be destroyed and Scan button is reenabled (it was disabled during the scanning process).

We should also handle the possibility of user closing the program by clicking the 'X' button while the background scanner is active. We must catch the `OnFormCloseQuery` event and tell the task to terminate.

```
procedure TfrmBackgroundFileSearchDemo.FormCloseQuery(Sender: TObject;
  var CanClose: boolean);
begin
  if assigned(FScanTask) then begin
    FScanTask.Terminate;
    FScanTask := nil;
    CanClose := true;
  end;
end;
```

The `Terminate` method will do two things – tell the task to terminate and then wait for its termination. After that, we simply have to clear the task reference and allow the program to terminate.

Let's move to the scanning part now. The `ScanFolders` method (which is the main task method, the one we passed to the `CreateTask`) splits the value of the `FolderMask` parameter into the folder and mask parts and passes them to the main worker method `ScanFolder`.

```
procedure ScanFolders(const task: IOmniTask);
var
  folder: string;
  mask  : string;
begin
  mask := task.ParamByName['FolderMask'];
  folder :=ExtractFilePath(mask);
  Delete(mask, 1, Length(folder));
  if folder <> '' then
    folder := IncludeTrailingPathDelimiter(folder);
  ScanFolder(task, folder, mask);
end;
```

`ScanFolder` first finds all subfolders of the selected folder and calls itself recursively for each subfolder. That means that we'll first process deepest folders and then proceed to the top of the folder tree.

Then it sends a message `MSG_SCAN_FOLDER` to the main thread. As a parameter of this message it sends the name of the folder being processed. There's nothing magical about this message – it is just an arbitrary numeric constant from range 0..65534 (number 65535 is reserved for internal OmniThreadLibrary use).

```
const
  MSG_SCAN_FOLDER  = 1;
  MSG_FOLDER_FILES = 2;

procedure ScanFolder(const task: IOmniTask; const folder, mask: string);
var
  err        : integer;
  folderFiles: TStringList;
  S          : TSearchRec;
begin
  err := FindFirst(folder + '*.*', faDirectory, S);
  if err = 0 then try
    repeat
      if ((S.Attr and faDirectory) <> 0) and (S.Name <> '.') and
         (S.Name <> '..')
      then
        ScanFolder(task, folder + S.Name + '\', mask);
      err := FindNext(S);
    until task.Terminated or (err <> 0);
  finally FindClose(S); end;
  task.Comm.Send(MSG_SCAN_FOLDER, folder);
  folderFiles := TStringList.Create;
  try
    err := FindFirst(folder + mask, 0, S);
    if err = 0 then try
      repeat
        folderFiles.Add(folder + S.Name);
        err := FindNext(S);
      until task.Terminated or (err <> 0);
    finally FindClose(S); end;
  finally task.Comm.Send(MSG_FOLDER_FILES, folderFiles); end;
end;
```

`ScanFolder` then runs the `FindFirst`/`FindNext`/`FindClose` loop for the second time to search for files in the folder. [BTW, if you want to first scan folders nearer to the root, just exchange the two loops and scan for files first and for folders second.] Each file is added to an internal `TStringList` object which was created just a moment before. When a folder scan is completed, this object is sent to the main thread as a parameter of the `MSG_FOLDER_FILES` message.

This approach – sending data for one folder – is a compromise between returning the complete set (full scanned tree), which would not provide a good feedback, and returning each file as we detect it, which would unnecessarily put a high load on the system.

Both `Find` loops test the state of the `task.Terminated` function and exit immediately if it is `True`. That allows us to terminate the background task when the user closes the application and the `OnFormCloseQuery` is called.

That's all that has to be done in the background task but we still have to process the messages in the main thread. For that, we have to implement the `OTLMonitor`'s `OnTaskMessage` event.

```
procedure TfrmBackgroundFileSearchDemo.OTLMonitorTaskMessage(
  const task: IOmniTaskControl);
var
  folderFiles: TStringList;
```

```
  msg         : TOmniMessage;
begin
  task.Comm.Receive(msg);
  if msg.MsgID = MSG_SCAN_FOLDER then
    FWaitingMessage := msg.MsgData
  else if msg.MsgID = MSG_FOLDER_FILES then begin
    folderFiles := TStringList(msg.MsgData.AsObject);
    FFileList.AddStrings(folderFiles);
    FreeAndNil(folderFiles);
    FWaitingCount := IntToStr(FFileList.Count);
  end;
end;
```

If the message is `MSG_SCAN_FOLDER` we just copy folder name to a local field. If the message is `MSG_FOLDER_FILES`, we copy file names from the parameter (which is a `TStringList`) to the global `FFileList` list and destroy the parameter. We also update a local field holding the number of currently found files.

Why don't we directly update two edit fields on the form (one with current folder and another with number of found files)? The background task can send many messages in one second (when processing folders with small number of files) and there's no point in displaying them all – the user will never see what was displayed anyway. It would also slow down the GUI because Windows controls would be updated hundreds of times per second, which is never a good idea.

Instead of that we just store the strings to be displayed in two form fields and display them from a timer which is triggered three times per second. That will not show all scanned folders and all intermediate file count results, but will still provide the user with the sufficient feedback.

```
procedure TfrmBackgroundFileSearchDemo.tmrDisplayStatusTimer(
  Sender: TObject);
begin
  if FWaitingMessage <> '' then begin
    outScanning.Text := FWaitingMessage;
    FWaitingMessage := '';
  end;
  if FWaitingCount <> '' then begin
    outFiles.Text := FWaitingCount;
    FWaitingCount := '';
  end;
end;
```

7.2 Web download and database storage

I need to download a list of web pages, extract data from them and then store this data in a SQLite database. The downloading/extracting part will happen in multiple threads (I'm using Synapse), but querying the database needs to be done asynchronously as I can only have one concurrent access to it.

The simplest approach is to create a Pipeline with two stages – multiple http retrievers in the first stage and one database writer in the second stage. The number of concurrent http retrievers would have be determined with some testing. It will depend on the throughput of the internet connection and on the quantity of the postprocessing done on the retrieved pages.

First pipeline stage, `Retriever`, fetches contents of one page. If the page is fetched correctly, a page description object (not shown in this demo) is created and sent to the output pipeline. Internally (not shown in this demo), `TPage.Create` could parse the page and extract the data.

As there can be at most one output generated for each input, this stage is implemented as a simple stage meaning that the Pipeline itself will loop over the input data and call the `Retriever` for each input.

Second stage, `Inserter`, is implemented as a normal stage (so it has to loop internally over all input data). First it establishes a connection to the database, then it loops over all input values (over data from all successfully retrieved pages) and inserts each result into the database. At the end (when there is no more data to process) it closes the database connection.

Main method (`ParallelWebRetriever`) first sets up and starts the pipeline. Next it feeds URLs to be retrieved into the input pipeline and marks the input pipeline as completed. At the end it waits for the pipeline to complete.

The program will execute as follows:

- `ParallelWebRetriever` starts the pipeline.
- OmniThreadLibrary sets up tasks for the pipeline; there will be twice as many first stage tasks as there are number of cores accessible from the process (this value was determined by guessing and would probably have to be adjusted in the real application) and only one task running the second stage.
- `ParallelWebRetriever` starts inserting URLs into the pipeline's input queue.
- First stage tasks immediately start processing these URLs and retrieving data from the web.
- At some moment, `ParallelWebRetriever` will run out of URLs and mark pipeline's input as completed.
- One by one, first stage tasks will finish processing the data. Each will send the data to the second stage over the pipeline and fetch new URL from the input queue.
- Second stage will read processed data item by item and write each item into the database. If it runs out of data to store (maybe the internet is slow today) it will wait for the next data item to appear in the pipeline.
- When a first stage task finishes a job and there's no more data in the pipeline's input queue, it will immediately exit because the pipeline's input is marked completed.
- After all first stage tasks exit (because of lack of the input data), Pipeline will detect this and mark the queue leading to the second stage as completed.
- After that, second stage will process all remaining items in its input queue and then exit.
- This will shut down the pipeline and `WaitFor` call will exit.

```
1 uses
2   OtlCommon,
3   OtlCollections,
4   OtlParallel;
5
6 function HttpGet(url: string; var page: string): boolean;
7 begin
8   // retrieve page contents from the url; return False if page is not accessible
9 end;
10
11 procedure Retriever(const input: TOmniValue; var output: TOmniValue);
12 var
13   pageContents: string;
14 begin
15   if HttpGet(input.AsString, pageContents) then
16     output := TPage.Create(input.AsString, pageContents);
17 end;
18
```

```
procedure Inserter(const input, output: IOmniBlockingCollection);
var
  page   : TOmniValue;
  pageObj: TPage;
begin
  // connect to database
  for page in input do begin
    pageObj := TPage(page.AsObject);
    // insert pageObj into database
    FreeAndNil(pageObj);
  end;
  // close database connection
end;

procedure ParallelWebRetriever;
var
  pipeline: IOmniPipeline;
  s       : string;
  urlList : TStringList;
begin
  // set up pipeline
  pipeline := Parallel.Pipeline
    .Stage(Retriever).NumTasks(Environment.Process.Affinity.Count * 2)
    .Stage(Inserter)
    .Run;
  // insert URLs to be retrieved
  for s in urlList do
    pipeline.Input.Add(s);
  pipeline.Input.CompleteAdding;
  // wait for pipeline to complete
  pipeline.WaitFor(INFINITE);
end;
```

7.3 Parallel for with synchronized output

I'm using the `Parallel.ForEach` method for parsing a file. The items in the file are bank statements with corresponding transactions.

I want to parse all the bank statements and add the resulting data to a general `TList<T>` containing all the parsed data.

I know that adding a `T` to a `TList<T>` is the problem but I have no idea how to solve it.

The best way is to use built-in capabilities of the For Each abstraction which allows you to write data to a shared blocking collection. Your program could then read data from this blocking collection and repack it to a `TList<T>`.

A solution to this problem is included with the OmniThreadLibrary distribution in folder `examples/forEach output`.

```
procedure ProcessTransactions(input: TStringList; output: TList<TTransaction>);
var
  outQueue   : IOmniBlockingCollection;
  transaction: TOmniValue;
begin
  outQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(0, input.Count - 1)
    .NoWait
    .PreserveOrder
    .Into(outQueue)
    .Execute(
      procedure(const value: integer; var result: TOmniValue)
      begin
        result := TTransaction.Create(input[value]);
      end
```

```
    );
  for transaction in outQueue do
    output.Add(transaction.AsObject as TTransaction);
end;
```

The code first creates a blocking collection that will 'pipe out' data from the For Each abstraction.

Next it starts a parallel for loop. It will iterate over all elements in the `input` list (`ForEach`), will preserve the order of the original items (`PreserveOrder`) and will write output into the blocking collection (`Into`). It will also run in background without waiting for all input to be processed (`NoWait`) so that the code in the main thread (`for transaction in`) can continue executing in parallel with the `ForEach`.

The parallel for worker code just creates a `TTransaction` object from the input line and stores it in the `result` variable. `ForEach` code will take this `result` and store it in the `outQueue`. If you don't want to produce a result for the given input value, just don't set the `result` variable.

This code also solves the stopping problem. The `for transaction in` loop will run until all of the input is processed. Only when the `ForEach` is truly finished, the `for transaction in` will exit, `ProcessTransaction` will also exit and the object running the parallel for loop will be automatically destroyed.

Below is the full code for a test program, implemented in a single form with a single component – ListBox1.

```
unit ForEachOutput1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls, Generics.Collections,
  OtlCommon,
  OtlCollections,
  OtlParallel;

type
  TTransaction = class
    Transaction: string;
    constructor Create(const transact: string);
  end;

  TfrmForEachOutput = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
  end;

var
  frmForEachOutput: TfrmForEachOutput;

implementation

{$R *.dfm}

procedure ProcessTransactions(input: TStringList;
  output: TList<TTransaction>);
var
  outQueue    : IOmniBlockingCollection;
  transaction: TOmniValue;
begin
  outQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(0, input.Count - 1)
    .NoWait
```

```
    .PreserveOrder
    .Into(outQueue)
    .Execute(
      procedure(const value: integer; var result: TOmniValue)
      begin
        result := TTransaction.Create(input[value]);
      end
    );
  for transaction in outQueue do
    output.Add(transaction.AsObject as TTransaction);
end;

procedure TfrmForEachOutput.FormCreate(Sender: TObject);
var
  bankStatements: TStringList;
  ch            : char;
  transaction   : TTransaction;
  transactions : TList<TTransaction>;
begin
  bankStatements := TStringList.Create;
  try
    for ch := '1' to '9' do bankStatements.Add(ch); //for testing
    transactions := TList<TTransaction>.Create;
    try
      ProcessTransactions(bankStatements, transactions);
      for transaction in transactions do
        ListBox1.Items.Add(transaction.Transaction);
    finally FreeAndNil(transactions); end;
  finally FreeAndNil(bankStatements); end;
end;

{ TTransaction }

constructor TTransaction.Create(const transact: string);
begin
  Transaction := transact;
end;

end.
```

If you don't need the output order to be preserved, you can also run the parallel for loop enumerating directly over the `input` container as in the following example:

```
procedure ProcessTransactions(input: TStringList;
  output: TList<TTransaction>);
var
  outQueue   : IOmniBlockingCollection;
  transaction: TOmniValue;
begin
  outQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach<string>(input).NoWait.Into(outQueue).Execute(
    procedure(const value: string; var result: TOmniValue)
    begin
      result := TTransaction.Create(value);
    end
  );
  for transaction in outQueue do
    output.Add(transaction.AsObject as TTransaction);
end;
```

7.4 Using taskIndex and task initializer in parallel for

I want to process some data in an array and generate aggregation result. For performance reasons I would like intermediate results for each worker to be stored in a separate part of memory and I want to merge those partial results at the end.

Can this be achieved by using `Parallel.For`?

This can be done by using the Parallel for `taskIndex` parameter. The example below also demonstrates the use of a task initializer which is strictly speaking not necessary in this case.

> A solution to this problem is included with the OmniThreadLibrary distribution in folder `examples/stringlist parser`.

The code below counts how many numbers in a big array of randomly generated data end in 0, 1, ... 9 and reports this result at the end. Each worker generates a partial result for a part of input array and results are merged at the end.

> This example is included with the OmniThreadLibrary distribution in demo `57_For`.

Let's assume we have a big array of test data (`testData: array of integer`). We can easily generate this data with a call to `Parallel.For`.

```
1 Parallel.For(Low(testData), High(testData)).Execute(
2   procedure (idx: integer)
3   begin
4     testData[idx] := Random(MaxInt);
5   end);
```

This data is not very random (but is still random enough for the purpose of this demo). The Parallel data production example shows a better way to generate random numbers in multiple worker threads.

As we have to prepare data storage for each worker thread, we have to know how many worker threads will be running. Therefore, we have to set the number of workers by calling `NumTasks`. A good default for a CPU intensive operation we'll be executing is to create one worker task for each available core.

```
1 type
2   TBucket = array [0..9] of integer;
3
4 var
5   buckets: array of TBucket;
6
7   numTasks := Environment.Process.Affinity.Count;
8   SetLength(buckets, numTasks);
```

Each `buckets` element will store data for one worker thread.

The for loop is next started with `numTasks` tasks. For each task an initializer (a parameter provided to the `.Initialize` call) is called with the appropriate `taskIndex` (from 0 to `numTasks - 1`). Initializer just sets the bucket that is associated with the task to zero. [This could easily be done in a main thread for all tasks at once, but I wanted to show how initializer can be used.]

Next, the `.Execute` is called and provided with a delegate which accepts two parameters – the task index `taskIndex` and the current value of the for loop `idx`. The code determines the last digit of the `testData[idx]` and increments the appropriate slot in the bucket that belongs to the current task.

```
1 Parallel.For(Low(testData), High(testData))
2   .NumTasks(numTasks)
3   .Initialize(
4     procedure (taskIndex, fromIndex, toIndex: integer)
5     begin
6       FillChar(buckets[taskIndex], SizeOf(TBucket), 0);
```

```
7     end)
8   .Execute(
9     procedure (taskIndex, idx: integer)
10    var
11      lastDigit: integer;
12    begin
13      lastDigit := testData[idx] mod 10;
14      buckets[taskIndex][lastDigit] := buckets[taskIndex][lastDigit] + 1;
15    end);
```

At the end, partial data is aggregated in the main thread. Result is stored in `buckets[0]`.

```
1 for j := 0 to 9 do begin
2   for i := 1 to numTasks - 1 do
3     buckets[0][j] := buckets[0][j] + buckets[i][j];
4 end;
```

7.5 Background worker and list partitioning

This is a simplification of my real-world task:

- Input is a string.
- Output is a `TStringList` containing characters (on per TStringList item) of the input string.

A background thread (master) grabs the input string and splits it into several pieces. For each piece it creates a new thread (child). Child thread receives its piece, splits it into characters and returns the result to the masterthread.

Main thread is sending workloads (strings) to the master thread. At any time the master thread could be signalled to terminate all child threads and itself.

When everything is done, the application ' s main thread processes the string list. Preferably the order of the characters should be the same as their original order.

Example:

- Input is string `'delphi'`.
- Master thread splits it into two parts - `'del'` and `'phi'`.
- Master thread starts two child threads. First receives input `'del'` and second input `'phi'`.
- First child thread splits `'del'` into three strings - `'d'`, `'e'`, `'l'` - and returns them to the master thread.
- Second child thread splits `'phi'` into three strings - `'p'`, `'h'`, `'i'` - and returns them to the master thread.
- Master thread returns a `TStringList` containing strings `'d'`, `'e'`, `'l'`, `'p'`, `'h'` and `'i'` to the main thread.

> You should keep in mind that this is really just a simplified example because there is no sense in splitting short strings into character in multiple threads. A solution to this problem is included with the OmniThreadLibrary distribution in folder `examples/stringlist parser`.

The solution below implements the master task (although the question mentioned threads I will describe the answer in the context of tasks) as a Background worker abstraction because it solves two problems automatically:

- Background worker accepts multiple work requests and pairs results with requests. The original question didn' t specify whether the master should accept more than
-

one request at the same time but I decided to stay on the safe side and allow this. Background worker supports work request cancellation.

The child tasks are implemented as a Paralel task abstraction. It allows us to run a code in multiple parallel tasks at the same time.

To set up a background worker, simply call `Parallel.BackgroundWorker` and provide it with a code that will process work items (`BreakStringHL`) and a code that will process results of the work item processor (`ShowResultHL`). It is important to keep in mind that the former (`BreakStringHL`) executes in the background thread while the latter (`ShowResultHL`) executes in the main thread. [Actually, it executes in the thread which calls `Parallel.BackgroundWorker` but in most cases that will be the main thread.]

```
FBackgroundWorker := Parallel.BackgroundWorker
  .Execute(BreakStringHL)
  .OnRequestDone(ShowResultHL);
```

Tearing it down is also simple.

```
FBackgroundWorker.CancelAll;
FBackgroundWorker.Terminate(INFINITE);
FBackgroundWorker := nil;
```

`CancellAll` is called to cancel any pending work requests, `Terminate` stops the worker (and waits for it to complete execution) and assignment clears the interface variable and destroys last pieces of the worker.

The `BreakStringHL` method takes a work item (which will contain the input string), sets up a parallel task abstraction, splits the input string into multiple strings and sends each one to the parallel task.

```
procedure TfrmStringListParser.BreakStringHL(
  const workItem: IOmniWorkItem);
var
  charsPerTask : integer;
  input        : string;
  iTask        : integer;
  numTasks     : integer;
  output       : TStringList;
  partialQueue : IOmniBlockingCollection;
  s            : string;
  stringBreaker: IOmniParallelTask;
  taskResults  : array of TStringList;
begin
  partialQueue := TOmniBlockingCollection.Create;
  numTasks := Environment.Process.Affinity.Count - 1;

  // create multiple TStringLists, one per child task
  SetLength(taskResults, numTasks);
  for iTask := Low(taskResults) to High(taskResults) do
    taskResults[iTask] := TStringList.Create;

  // start child tasks
  stringBreaker := Parallel.ParallelTask.NumTasks(numTasks).NoWait
    .TaskConfig(
      Parallel.TaskConfig.CancelWith(workItem.CancellationToken))
    .Execute(
      procedure (const task: IOmniTask)
      var
        workItem: TOmniValue;
      begin
        workItem := partialQueue.Next;
        SplitPartialList(workItem[1].AsString,
          taskResults[workItem[0].AsInteger], task.CancellationToken);
```

```
      end
    );

  // provide input to child tasks
  input := workItem.Data;
  for iTask := 1 to numTasks do begin
    // divide the remaining part in as-equal-as-possible segments
    charsPerTask := Round(Length(input) / (numTasks - iTask + 1));
    partialQueue.Add(TOmniValue.Create([iTask-1,
      Copy(input, 1, charsPerTask)]));
    Delete(input, 1, charsPerTask);
  end;

  // process output
  stringBreaker.WaitFor(INFINITE);
  if not workItem.CancellationToken.IsSignalled then begin
    output := TStringList.Create;
    for iTask := Low(taskResults) to High(taskResults) do begin
      for s in taskResults[iTask] do
        output.Add(s);
    end;
    workItem.Result := output;
  end;
  for iTask := Low(taskResults) to High(taskResults) do
  taskResults[iTask].Free;
end;
```

`BreakStringHL` is called for each input string that arrives over the communication channel. It firstly decides how many threads to use (number of cores minus one; the assumption here is that one core is used to run the main thread). One string list is then created for each child subtask. It will contain the results generated from that task.

A Parallel task abstraction is then started, running (number of cores minus one) tasks. Each will accept a work unit on an internally created queue, process it and shut down.

Next, the code sends work units to child tasks. Each work unit contains the index of the task (so that it knows where to store the data) and the string to be processed. All child tasks also get the same cancellation token so that they can be cancelled in one go. Child tasks are executed in a thread pool to minimize thread creation overhead.

When all child tasks are completed, partial results are collected into one `TStringList` object which is returned as a result of the background worker work item.

Actual string breaking is implemented as a standalone procedure. It checks each input character and signals the cancelation token if the character is an exclamation mark. (This is implemented just as a cancelation testing mechanism.) It exits if the cancelation token is signalled. At the end, `Sleep(100)` simulates heavy processing and allows the user to click the Cancel button in the GUI before the operation is completed.

```
procedure SplitPartialList(const input: string; output: TStringList;
  const cancel: IOmniCancellationToken);
var
  ch: char;
begin
  for ch in input do begin
    if ch = '!' then // for testing
      cancel.Signal;
    if cancel.IsSignalled then
      break; //for ch
    output.Add(ch);
    Sleep(100); // simulate workload
  end;
end;
```

The example program uses simple `OnClick` handler to send string to processing.

```
procedure TfrmStringListParser.btnProcessHLClick(Sender: TObject);
begin
  FBackgroundWorker.Schedule(
    FBackgroundWorker.CreateWorkItem(inpString.Text));
end;
```

Results are returned to the `ShowResultHL` method (as it was passed as a parameter to the `OnRequestDone` call when creating the background worker).

```
procedure TfrmStringListParser.ShowResultHL(
  const Sender: IOmniBackgroundWorker;
  const workItem: IOmniWorkItem);
begin
  if workItem.CancellationToken.IsSignalled then
    lbLog.Items.Add('Canceled')
  else
    ShowResult(workItem.Result.AsObject as TStringList);
end;
```

It receives an `IOmniBackgroundWorker` interface (useful if you are sharing one method between several background workers) and the work item that was processed (or cancelled). The code simply checks if the work item was cancelled and displays the result (by using the `ShowResult` from the original code) otherwise.

The demonstration program also implements a Cancel button which cancels all pending operations.

```
procedure TfrmStringListParser.btnCancelHLClick(Sender: TObject);
begin
  FBackgroundWorker.CancelAll;
end;
```

All not-yet-executing operations will be cancelled automatically. For the string that is currently being processed, a cancellation token will be signalled. `SplitPartialList` will notice this token being signalled and will stop processing.

7.6 Parallel data production

> This question comes from StackOverflow. It is reproduced here in a slightly shortened form.

I am looking into generating a file (750 MB) full of random bytes. The problem is that it takes ages until the process completes. Any ideas for a faster approach?

This solution uses Parallel Task abstraction.

The algorithm works as follows:

- do in parallel:
  - repeat
    - find out how many bytes to process in this iteration
      - if there's no more work to do, exit the loop
    - prepare the buffer
    - send it to the output queue

The tricky part is implementing the third item – 'find out how many bytes to process in this iteration' – in a lock-free fashion. What we need is a thread-safe equivalent of the following (completely thread-unsafe) fragment.

```
if fileSize > CBlockSize then
  numBytes := CBlockSize
else
  numBytes := fileSize;
fileSize := fileSize - numBytes;
```

OmniThreadLibrary implements a thread-safe version of this pattern in TOmniCounter.Take. If you have TOmniCounter initialized with some value (say, fileSize) and you call TOmniCounter.Take(numBytes), the code will behave exactly the same as the fragment above except that it will work correctly if Take is called from multiple threads at the same time. In addition to that, the new value of the fileSize will be stored in the TOmniCounter's counter and returned as a function result.

There's another version of Take which returns the result in a var parameter and sets its result to True if value returned is larger than zero.

```
function TOmniCounterImpl.Take(count: integer;
  var taken: integer): boolean;
begin
  taken := Take(count);
  Result := (taken > 0);
end; { TOmniCounterImpl.Take }
```

This version of Take allows you to write elegant iteration code which also works when multiple tasks are accessing the same counter instance.

```
counter := CreateCounter(numBytes);
while counter.Take(blockSize, blockBytes) do begin
  // process blockBytes bytes
end;
```

The solution creates a counter which holds number of bytes to be generated (unwritten) and a queue (outQueue) that will hold generated data buffers until they are written to a file. Then it starts a ParallelTask abstraction on all available cores. While the abstraction is running in the background (because NoWait is used), the main thread continues with the CreateRandomFile execution, reads the data from the outQueue and writes blocks to the file.

```
procedure CreateRandomFile(fileSize: integer; output: TStream);
const
  CBlockSize = 1 * 1024 * 1024 {1 MB};
var
  buffer   : TOmniValue;
  memStr   : TMemoryStream;
  outQueue : IOmniBlockingCollection;
  unwritten: IOmniCounter;
begin
  outQueue := TOmniBlockingCollection.Create;
  unwritten := CreateCounter(fileSize);
  Parallel.ParallelTask.NoWait
    .NumTasks(Environment.Process.Affinity.Count)
    .OnStop(Parallel.CompleteQueue(outQueue))
    .Execute(
      procedure
      var
        buffer      : TMemoryStream;
        bytesToWrite: integer;
        randomGen   : TGpRandom;
      begin
        randomGen := TGpRandom.Create;
        try
          while unwritten.Take(CBlockSize, bytesToWrite) do begin
            buffer := TMemoryStream.Create;
            buffer.Size := bytesToWrite;
            FillBuffer(buffer.Memory, bytesToWrite, randomGen);
```

```
28              outQueue.Add(buffer);
29            end;
30          finally FreeAndNil(randomGen); end;
31        end
32      );
33    for buffer in outQueue do begin
34      memStr := buffer.AsObject as TMemoryStream;
35      output.CopyFrom(memStr, 0);
36      FreeAndNil(memStr);
37    end;
38  end;
```

The parallel part firstly creates a random generator in each tasks. Because the random generator code is not thread-safe, it cannot be shared between the tasks. Next it uses the above-mentioned `Take` pattern to grab a bunch of work, generates that much random data (inside the `FillBuffer` which is not shown here) and adds the buffer to the `outQueue`.

You may be asking yourself how will this code stop? When the `unwritten` counter drops to zero, `Take` will fail in every task and anonymous method running inside the task will exit. When this happens in all tasks, `OnStop` handler will be called automatically.

The code above passes `Parallel.CompleteQueue` to the `OnStop`. This is a special helper which creates a delegate that calls `CompleteAdding` on its parameter. Therefore, `OnStop` handler will call `outQueue.CompleteAdding`, which will cause the `for` loop in `CreateRandomFile` to exit after all data is processed.

7.7 Building a connection pool

Is it possible to use OTL to create Connection Pool systems? It seems like OTL solves a lot of the concurrency and communications issues. Is this feasible?

The thread pool enables connection pooling by providing property `ThreadData: IOtlThreadData` to each task. This property is bound to a thread – it is created when a thread is created and is destroyed together with the thread.

To facilitate this, task implements property `ThreadData` which contains the user data associated with the thread.

```
1 type
2   IOmniTask = interface
3     ...
4     property ThreadData: IInterface;
5   end;
```

This data is initialized in the thread pool when a new thread is created. It is destroyed automatically when a thread is destroyed.

To initialize the `ThreadData`, you have to write a 'factory' method, a method that creates a thread data interface. The thread pool will call this factory method to create the thread data and will then assign the same object to all tasks running in that thread.

```
1 type
2   TOTPThreadDataFactoryFunction = function: IInterface;
3   TOTPThreadDataFactoryMethod = function: IInterface of object;
4
5   IOmniThreadPool = interface
6     ...
7     procedure SetThreadDataFactory(
8       const value: TOTPThreadDataFactoryMethod); overload;
9     procedure SetThreadDataFactory(
10      const value: TOTPThreadDataFactoryFunction); overload;
11  end;
```

You can write two kinds of a thread data factories – a 'normal' function that returns an `IInterface` or a method function (a function that belongs to a class) that returns an `IInterface`.

7.7.1 From theory to practice

Let's return to the practical part. In the database connection pool scenario, you'd have to write a connection interface, object and factory (see demo application `24_ConnectionPool` for the full code).

In the `OnCreate` event the code creates a thread pool, assigns it a name and thread data factory. The latter is a function that will create and initialize new connection for each new thread. In the `OnClose` event the code terminates all waiting tasks (if any), allowing the application to shutdown gracefully. `FConnectionPool` is an interface and its lifetime is managed automatically so we don't have to do anything explicit with it.

```
procedure TfrmConnectionPoolDemo.FormCreate(Sender: TObject);
begin
  FConnectionPool := CreateThreadPool('Connection pool');
  FConnectionPool.SetThreadDataFactory(CreateThreadData);
  FConnectionPool.MaxExecuting := 3;
end;
```

Thread data factory could also be assigned to the global thread pool by calling `GlobalOmniThreadPool.SetThreadDataFactory`.

```
procedure TfrmConnectionPoolDemo.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  FConnectionPool.CancelAll;
end;
```

The magic `CreateThreadData` factory just creates a connection object (which would in a real program establish a database connection, for example).

```
function TfrmConnectionPoolDemo.CreateThreadData: IInterface;
begin
  Result := TConnectionPoolData.Create;
end;
```

There's no black magic behind this connection object. It is an object that implements an interface. Any interface. This interface will be used only in your code. In this demo, `TConnectionPoolData` contains only one field – unique ID, which will help us follow the program execution.

```
type
  IConnectionPoolData = interface
    function ConnectionID: integer;
  end;

  TConnectionPoolData = class(TInterfacedObject, IConnectionPoolData)
  strict private
    cpID: integer;
  public
    constructor Create;
    destructor  Destroy; override;
    function ConnectionID: integer;
  end;
```

As this is not a code from a real world application, I didn't bother connecting it to any specific database. `TConnectionPoolData` constructor will just notify the main form that it has begun its job, generate new ID and sleep for five seconds (to emulate establishing a slow connection). The destructor is even simpler, it just sends a notification to the main form.

```
constructor TConnectionPoolData.Create;
begin
  PostToForm(WM_USER, MSG_CREATING_CONNECTION,
    integer(GetCurrentThreadID));
  cpID := GConnPoolID.Increment;
  Sleep(5000);
  PostToForm(WM_USER, MSG_CREATED_CONNECTION, cpID);
end;

destructor TConnectionPoolData.Destroy;
begin
  PostToForm(WM_USER, MSG_DESTROY_CONNECTION, cpID);
end;
```

Creating and running a task is really simple with the OmniThreadLibrary.

```
procedure TfrmConnectionPoolDemo.btnScheduleClick(Sender: TObject);
begin
  Log('Creating task');
  CreateTask(TaskProc).MonitorWith(OTLMonitor).Schedule(FConnectionPool);
end;
```

We are monitoring the task with the `TOmniEventMonitor` component because a) we want to know when the task will terminate and b) otherwise we would have to store into a global field a reference to the `IOmniTaskControl` interface returned from the `CreateTask`.

The task worker procedure `TaskProc` is again really simple. First it pulls the connection data from the task interface (`task.ThreadData as IConnectionPoolData`), retrieves the connection ID and sends task and connection ID to the main form (for logging purposes) and then it sleeps for three seconds, indicating some heavy database activity.

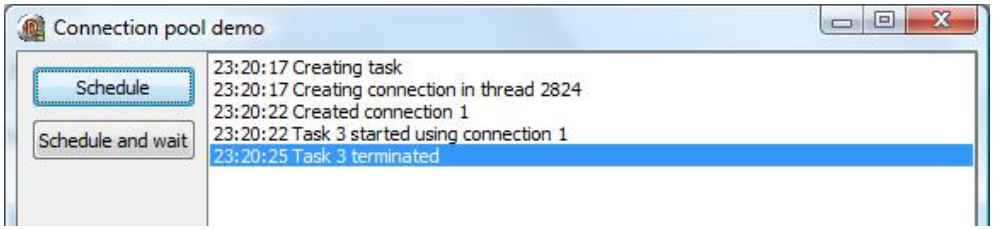
```
procedure TaskProc(const task: IOmniTask);
begin
  PostToForm(WM_USER + 1, task.UniqueID,
    (task.ThreadData as IConnectionPoolData).ConnectionID);
  Sleep(3000);
end;
```

Then ... but wait! There's no more! Believe it or not, that's all. OK, there is some infrastructure code that is used only for logging but that you can look up by yourself.

There is also a code assigned to the second button (Schedule and wait) but it only demonstrates how you can schedule a task and wait on its execution. This is useful if you're running the task from a background thread.

7.7.2 Running the demo

Let's run the demo and click on the Schedule key.



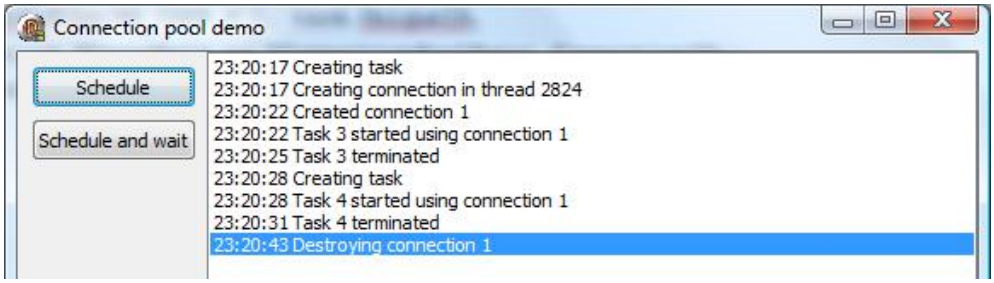
What happened here?

- Task was created.
- It was immediately scheduled for execution and thread pool called our thread data factory.
- Thread data object constructor waited for five seconds and returned.

Thread pool immediately started executing the task.

- Task waited for three seconds and exited.

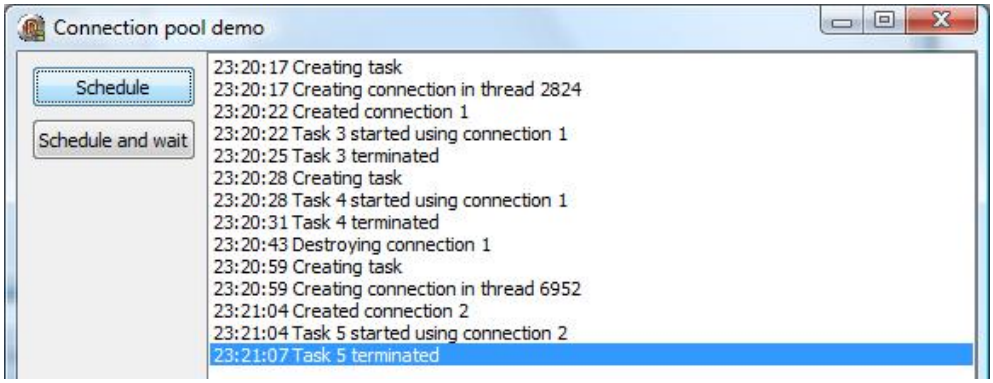
OK, nothing special. Let's click the Schedule button again.



Now a new task was created (with ID 4), was scheduled for execution in the same thread as the previous task and reused the connection that was created when the first task was scheduled. There is no 5 second wait, just the 3 second wait implemented in the task worker procedure.

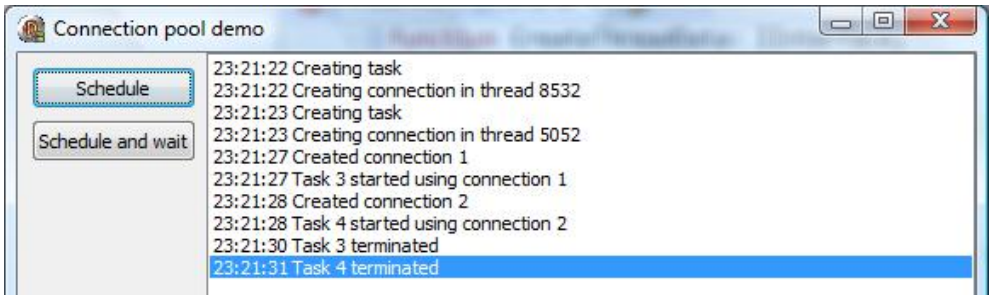
If you now let the program run for 10 seconds, a message 'Destroying connection 1' will appear. The reason for this is that the default thread idle timeout in a thread pool is 10 seconds. In other words, if a thread does nothing for 10 seconds, it will be stopped. You are, of course, free to set this value to any number or even to 0, which would disable the idle thread termination mechanism.

If you now click the Schedule button again, new thread will be created in the thread pool and new connection will be created in our factory function (spending 5 seconds doing nothing).

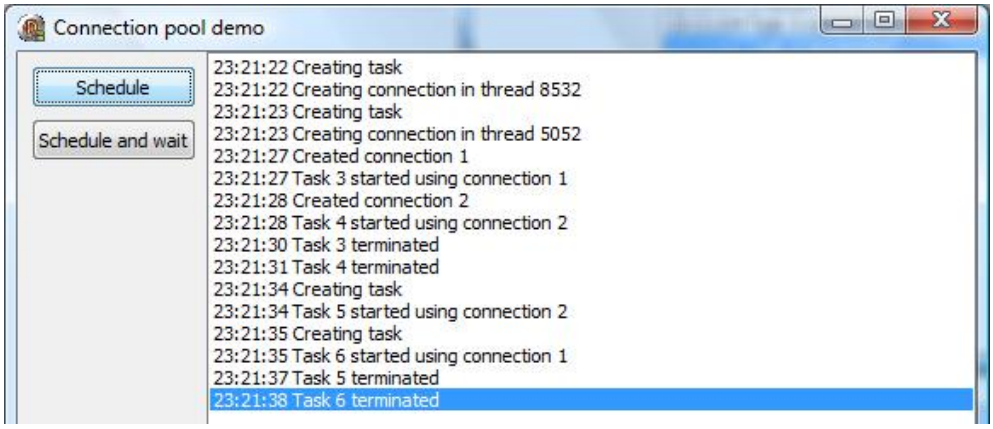


Let's try something else. I was running the demo on my laptop with a dual core CPU, which caused the thread pool to limit maximum number of currently executing threads to two. By default, thread pool uses as much threads as there are cores in the system, but again you can override the value. (In releases up to [3.03], you could use at most 60 currently executing threads. Starting from release [3.04], this number is only limited by the system resources.)

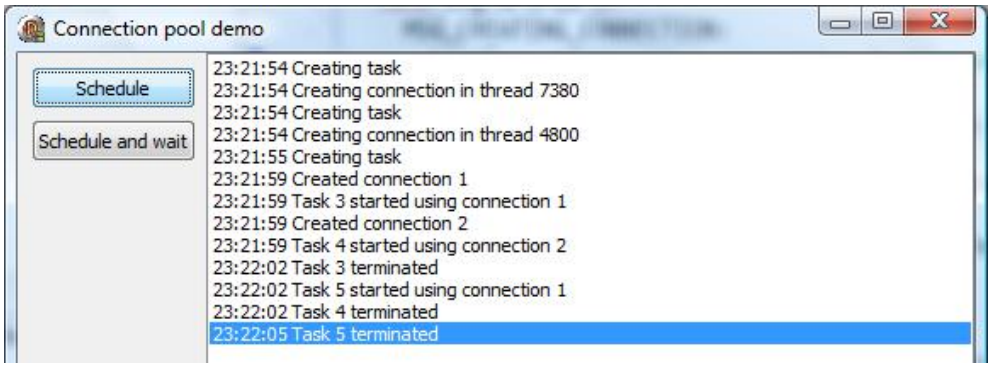
To recap – when running the demo, the thread pool was limited to two concurrent threads. When I clicked the Schedule button two times in a quick succession, first task was scheduled and first connection started being established (has entered the `Sleep` function). Then the second task was created (as the connection is being established from the worker thread, GUI is not blocked) and second connection started being established in the second thread. Five seconds later, connections are created and tasks start running (and wait three seconds, and exit).



Then I clicked the Schedule button two more times. Two tasks were scheduled and they immediately started execution in two worker threads.



For the third demo, I restarted the app and clicked the Shedule button three times. Only two worker threads were created and two connections established and two tasks started execution. The third task entered the thread pool queue and waited for the first task to terminate, after which it was immediately scheduled.



So here you have it – a very simple way to build a connection pool.

7.8 QuickSort and parallel max

I would like to sort a big array of data, but my comparison function is quite convoluted and sorting takes a long time. Can I use OmniThreadLibrary to speed up sorting?

On a similar topic – sometimes I'd also like to find a maximum data element in this big array, without doing the sorting. How would I approach this problem?

The answer to both parts of the problem is the same – use the Fork/Join abstraction.

7.8.1 QuickSort

The first part of this how-to implements a well-known quicksort algorithm in a parallel way (see demo application `44_Fork-Join QuickSort` for the full code).

Let's start with a non-optimized single threaded sorter. This simple implementation is very easy to convert to the multithreaded form.

```
procedure TSequentialSorter.QuickSort(left, right: integer);
var
  pivotIndex: integer;
begin
  if right > left then begin
    if (right - left) <= CSortThreshold then
      InsertionSort(left, right)
    else begin
      pivotIndex := Partition(left, right, (left + right) div 2);
      QuickSort(left, pivotIndex - 1);
      QuickSort(pivotIndex + 1, right);
    end;
  end;
end;
```

As you can see, the code switches to an insertion sort when the dimension of the array drops below some threshold. This is not really important for the single threaded version (it only brings a small speedup) but it will help immensely with the multithreaded version.

Converting this quicksort to a multithreaded version is quite simple.

Firstly, we have to create a Fork/Join computation pool. In this example, it is stored in a global field.

```
FForkJoin := Parallel.ForkJoin;
```

Secondly, we have to adapt the `QuickSort` method.

```
procedure TParallelSorter.QuickSort(left, right: integer);
var
  pivotIndex: integer;
  sortLeft  : IOmniCompute;
  sortRight : IOmniCompute;
begin
  if right > left then begin
    if (right - left) <= CSortThreshold then
      InsertionSort(left, right)
    else begin
      pivotIndex := Partition(left, right, (left + right) div 2);
      sortLeft := FForkJoin.Compute(
        procedure
        begin
          QuickSort(left, pivotIndex - 1);
        end);
      sortRight := FForkJoin.Compute(
        procedure
        begin
          QuickSort(pivotIndex + 1, right);
        end);
      sortLeft.Await;
      sortRight.Await;
    end;
  end;
end;
```

The code looks much longer but changes are really simple. Each recursive call to `QuickSort` is replaced with the call to `Compute` ...

```
sortLeft := FForkJoin.Compute(
  procedure
  begin
    QuickSort(left, pivotIndex - 1);
  end);
```

... and the code `Await`s on both subtasks.

Instead of calling `QuickSort` directly, parallel version creates `IOmniCompute` interface by calling `FForkJoin.Compute`. This creates a subtask wrapping the anonymous function which was passed to the `Compute` and puts this subtask into the Fork/Join computation pool.

The subtask is later read from this pool by one of the Fork/Join workers and is processed in the background thread.

Calling `Await` checks if the subtask has finished its work. In that case, `Await` simply returns and the code can proceed. Otherwise (subtask is still working), `Await` tries to get one subtask from the computation pool, executes it, and then repeats from the beginning (by checking if the subtask has finished its work). This way, all threads are always busy either with executing their own code or a subtask from the computation pool.

Because two `IOmniCompute` interfaces are stored on the stack in each `QuickSort` call, this code uses more stack space than the single threaded version. That is the main reason why the parallel execution is stopped at some level and simple sequential version is used to sort remaining fields.

7.8.2 Parallel max

The second part of this how-to finds a maximum element of an array in a parallel way (see demo application `45_Fork-Join max` for the full code).

The parallel solution is similar to the quicksort example above with few important differences related to the fact that the code must return a value (the quicksort code merely sorted the array returning nothing).

This directly affects the interface usage – instead of working with `IOmniForkJoin` and `IOmniCompute` the code uses `IOmniForkJoin<T>` and `IOmniCompute<T>`. As our example array contains integers, the parallel code creates `IOmniForkJoin<integer>` and passes it to the `ParallelMax` function.

```
1 max := ParallelMax(Parallel.ForkJoin<integer>, Low(FData), High(FData));
```

In this example Fork/Join computation pool is passed as a parameter. This approach is more flexible but is also slightly slower and – more importantly – uses more stack space.

```
1  function ParallelMax(
2    const forkJoin: IOmniForkJoin<integer>;
3    left, right: integer): integer;
4
5  var
6    computeLeft :IOmniCompute<integer>;
7    computeRight:  IOmniCompute<integer>;
8    mid         : integer;
9
10   function Compute(left, right: integer): IOmniCompute<integer>;
11   begin
12     Result := forkJoin.Compute(
13       function: integer
14       begin
15         Result := ParallelMax(forkJoin, left, right);
16       end
17     );
18   end;
19
20 begin
21   if (right - left) < CSeqThreshold then
22     Result := SequentialMax(left, right)
23   else begin
```

```
24     mid := (left + right) div 2;
25     computeLeft := Compute(left, mid);
26     computeRight := Compute(mid + 1, right);
27     Result := Max(computeLeft.Value, computeRight.Value);
28   end;
29 end;
```

When the array subrange is small enough, `ParallelMax` calls the sequential (single threaded) version – just as the parallel QuickSort did, and because of the same reason – not to run out of stack space.

With a big subrange, the code creates two `IOmniCompute<integer>` subtasks each wrapping a function returning an `integer`. This function in turn calls back `ParallelMax` (but with a smaller range). To get the result of the anonymous function wrapped by the `Compute`, the code calls the `Value` function. Just as with the `Await`, `Value` either returns a result (if it was already computed) or executes other Fork/Join subtasks from the computation pool.

While creating Fork/Join programs, keep in mind this anti-pattern. The following code fragment is wrong!

```
1 Result := Max(Compute(left, mid).Value,
2  Compute(mid + 1, right).Value);
```

You must always create all subtasks before calling `Await` or `Value`! Otherwise, your code will not execute in parallel at all – it will all be processed by a single thread.

7.9 Parallel search in a tree

My program often searches for data in a large tree structure. The comparison function is pretty convoluted and the search takes a lot of time. Could it be sped up by using parallelization?

The For Each abstraction can be used to iterate over complicated structures such as trees. The biggest problem is to assure that the code will always stop. The solution below achieves this by using special features of the blocking collection.

The solution to this problem is available as a demo application 35_ParallelFor.

The code in the demo application creates a big tree of `TNode` nodes. Each node contains a value (`Value`) and a list of child nodes (`Child`). `TNode` also implements a function to return the number of children (`NumChild`), a function that converts the node into a textual representation (`ToString`; used for printing out the result) and an enumerator that will allow us to access child nodes in a nice structured fashion (`Children`). To learn more about the implementation of `TNode` and its enumerator, see the demo program.

```
1 type
2   Node = class
3     Value: integer;
4     Child: array of TNode;
5     function  NumChild: integer;
6     function  ToString: string; reintroduce;
7     function  Children: TNodeChildEnumeratorFactory;
8   end;
```

For comparison purposes the demo program implements sequential search function `SeqScan` which uses recursion to traverse the tree.

```
1 function TfrmParallelForDemo.SeqScan(node: TNode;
2   value: integer): TNode;
3 var
```

```
  iNode: integer;
begin
  if node.Value = value then
    Result := node
  else begin
    Result := nil;
    for iNode := 0 to node.NumChild - 1 do begin
      Result := SeqScan(node.Child[iNode], value);
      if assigned(Result) then
        break; //for iNode
    end;
  end;
end;
```

The parallel version of this function is more complicated. It uses a blocking collection that is shared between all `ForEach` tasks. This blocking collection contains all nodes that have yet to be traversed. At the beginning, it contains only the root node. Each task executes the following pseudocode:

```
while <there are nodes in the blocking collection>
  <take one node from the blocking collection>
  <if the node contains the value we're searching for, stop>
  <put all children of this node into the blocking collection>
```

The real code is more complicated because of two complications. Firstly, when a value is found, all `ForEach` tasks must stop, not just the one that had found the value. Secondly, the code must stop if the value we're searching for is not present in the tree. In the pseudocode above this is automatically achieved by the condition in the `while` statement but in reality this is not so easy. At some time the blocking collection may be completely empty when there is still data to be processed. (For example just at the beginning when the first task takes out the root node of the tree. Yes, this does mean that the condition in the `while` statement above is not completely valid.)

```
function TfrmParallelForDemo.ParaScan(rootNode: TNode; value: integer): TNode;
var
  cancelToken: IOmniCancellationToken;
  nodeQueue : IOmniBlockingCollection;
  nodeResult : TNode;
  numTasks   : integer;
begin
  nodeResult := nil;
  cancelToken := CreateOmniCancellationToken;
  numTasks := Environment.Process.Affinity.Count;
  nodeQueue := TOmniBlockingCollection.Create(numTasks);
  nodeQueue.Add(rootNode);
  Parallel.ForEach(nodeQueue as IOmniValueEnumerable)
    .NumTasks(numTasks) // must be same number of task as in
                        // nodeQueue to ensure stopping
    .CancelWith(cancelToken)
    .Execute(
      procedure (const elem: TOmniValue)
      var
        childNode: TNode;
        node     : TNode;
      begin
        node := TNode(elem.AsObject);
        if node.Value = value then begin
          nodeResult := node;
          nodeQueue.CompleteAdding;
          cancelToken.Signal;
        end
        else for childNode in node.Children do
          nodeQueue.TryAdd(childNode);
      end);
  Result := nodeResult;
end;
```

The code first creates a cancellation token which will be used to stop the ForEach loop. Number of tasks is set to number of cores accessible from the process and a blocking collection is created.

Resource count for this collection is initialized to the number of tasks (numTasks parameter to the TOmniBlockingCollection.Create). This assures that the blocking collection will be automatically put into the 'completed' mode (as if the CompleteAdding had been called) if numTasks threads are simultaneously calling Take and the collection is empty. This prevents the 'resource exhaustion' scenario – if all workers are waiting for new data and the collection is empty, then there's no way for new data to appear and the waiting is stopped by putting the collection into completed state.

The root node of the tree is added to the blocking collection. Then the Parallel.ForEach is called, enumerating the blocking collection.

The code also passes cancellation token to the ForEach loop and starts the parallel execution. In each parallel task, the following code is executed (this code is copied from the full ParaScan example above):

```
1 procedure (const elem: TOmniValue)
2 var
3   childNode: TNode;
4   node     : TNode;
5 begin
6   node := TNode(elem.AsObject);
7   if node.Value = value then begin
8     nodeResult := node;
9     nodeQueue.CompleteAdding;
10    cancelToken.Signal;
11  end
12  else for childNode in node.Children do
13    nodeQueue.TryAdd(childNode);
14 end
```

The code is provided with one element from the blocking collection at a time. If the Value field is the value we're searching for, nodeResult is set, blocking collection is put into CompleteAdding state (so that enumerators in other tasks will terminate blocking wait (if any)) and cancellation token is signalled to stop other tasks that are not blocked.

Otherwise (not the value we're looking for), all the children of the current node are added to the blocking collection. TryAdd is used (and its return value ignored) because another thread may call CompleteAdding while the for childNode loop is being executed.

Parallel for loop is therefore iterating over a blocking collection into which nodes are put (via the for childNode loop) and from which they are removed (via the ForEach implementation). If child nodes are not provided fast enough, blocking collection will block on Take and one or more tasks may sleep for some time until new values appear. Only when the value is found, the blocking collection and ForEach loop are completed/cancelled.

7.10 Multiple workers with multiple frames

I am running multiple background tasks implemented with OmniThreadLibrary's CreateTask function and they are all interacting with the same form in the program. I want to separate user interface into separate frames, each interacting with one task but I can't find a good way to do it.

The solution to this problem can be split into three parts – the worker, the frame and the binding code in the form unit.

The solution to this problem is available as a demo application 49_FramedWorkers.

7.10.1 The worker

In this example (unit `test_49_Worker` in the demo application), the worker code is intentionally very simple. It implements a timer which, triggered approximately every second, sends a message to the owner. This message is received in a `Msg` parameter when the task is created. The worker can also respond to a `MSG_HELLO` message with a `'Hello'` response.

```
type
  TFramedWorker = class(TOmniWorker)
  strict private
    FMessage: string;
  public
    function Initialize: boolean; override;
    procedure MsgHello(var msg: TOmniMessage); message MSG_HELLO;
    procedure Timer1;
  end;

function TFramedWorker.Initialize: boolean;
begin
  Result := inherited Initialize;
  if Result then begin
    FMessage := Task.Param['Msg'];
    Task.SetTimer(1, 1000 + Random(500), @TFramedWorker.Timer1);
  end;
end;

procedure TFramedWorker.MsgHello(var msg: TOmniMessage);
begin
  Task.Comm.Send(MSG_NOTIFY, 'Hello, ' + msg.MsgData);
end;

procedure TFramedWorker.Timer1;
begin
  Task.Comm.Send(MSG_NOTIFY, '... ' + FMessage);
end;
```

Message ID's (`MSG_HELLO`, `MSG_NOTIFY`) are defined in unit `test_49_Common` as they are shared with the frame implementation.

7.10.2 The frame

The frame (unit `test_49_FrameWithWorker`) contains a listbox and a button. It implements a response function for the `MSG_NOTIFY` message – `MsgNotify` – and it contains a reference to the worker task. This reference will be set in the main form when the task and the frame are constructed.

```
type
  TfrmFrameWithWorker = class(TFrame)
    lbLog: TListBox;
    btnHello: TButton;
    procedure btnHelloClick(Sender: TObject);
  private
    FWorker: IOmniTaskControl;
  public
    property Worker: IOmniTaskControl read FWorker write FWorker;
    procedure MsgNotify(var msg: TOmniMessage); message MSG_NOTIFY;
  end;
```

The `MsgNotify` method is automatically called whenever the `MSG_NOTIFY` message is received by the frame. It merely shows the message contents.

```
procedure TfrmFrameWithWorker.MsgNotify(var msg: TOmniMessage);
```

```
2 begin
3   lbLog.ItemIndex := lbLog.Items.Add(msg.MsgData);
4 end;
```

A click on the button sends a `MSG_HELLO` message to the worker. A name of the frame is sent as a parameter. The worker will include this name in the response so that we can verify that the response is indeed sent to the correct frame.

```
1 procedure TfrmFrameWithWorker.btnHelloClick(Sender: TObject);
2 begin
3   Worker.Comm.Send(MSG_HELLO, Name);
4 end;
```

7.10.3 The form

Five frame/worker pairs are created in the form while it is being created. The code in `FormCreate` creates and positions each frame and then creates a worker named `Frame #%d` (where `%d` is replaced with the sequential number of the frame). Workers are created in the `CreateWorker` method.

```
1 const
2   CNumFrames = 5;
3   CFrameWidth = 150;
4   CFrameHeight = 200;
5
6 function TfrmFramedWorkers.CreateFrame(left, top, width, height: integer;
7   const name: string): TfrmFrameWithWorker;
8 begin
9   Result := TfrmFrameWithWorker.Create(Self);
10  Result.Parent := Self;
11  Result.Left := left;
12  Result.Top := top;
13  Result.Width := width;
14  Result.Height := height;
15  Result.Name := name;
16 end;
17
18 procedure TfrmFramedWorkers.FormCreate(Sender: TObject);
19 var
20   frame : TfrmFrameWithWorker;
21   iFrame: integer;
22 begin
23   FTaskGroup := CreateTaskGroup;
24   for iFrame := 1 to CNumFrames do begin
25     frame := CreateFrame(
26       CFrameWidth * (iFrame - 1), 0, CFrameWidth, CFrameHeight,
27       Format('Frame%d', [iFrame]));
28     CreateWorker(frame, Format('Frame #%d', [iFrame]));
29   end;
30   ClientWidth := CNumFrames * CFrameWidth;
31   ClientHeight := CFrameHeight;
32 end;
```

The `FormCreate` method also creates a task group which is used to terminate all workers when a form is closed.

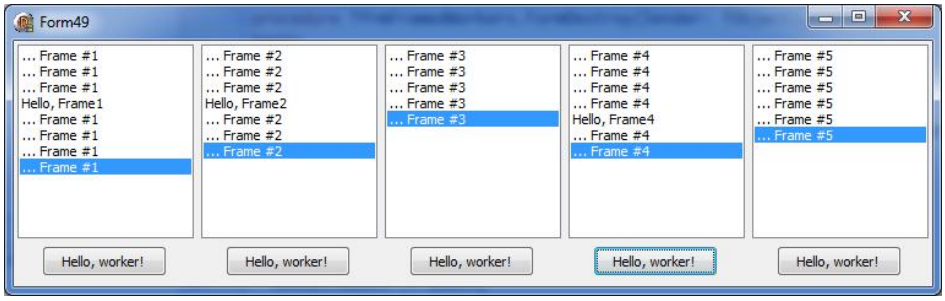
```
1 procedure TfrmFramedWorkers.FormDestroy(Sender: TObject);
2 begin
3   FTaskGroup.TerminateAll;
4 end;
```

The final piece of the puzzle is the `CreateWorker` method. It creates a low-level task and sets its name. The same name is assigned to the `Msg` parameter so that it will be used in messages sent from the task. The `OnMessage` call assigns the frame to function as a 'message-processor' for the tasks – all messages from the task will be dispatched to the

frame. That's how the `MSG_NOTIFY` message ends up being processed by the frame's `MsgNotify` method.

```
procedure TfrmFramedWorkers.CreateWorker(frame: TfrmFrameWithWorker;
  const caption: string);
var
  worker: IOmniTaskControl;
begin
  worker := CreateTask(TFramedWorker.Create(), caption)
    .SetParameter('Msg', caption)
    .OnMessage(frame)
    .Run;
  frame.Worker := worker;
  FTaskGroup.Add(worker);
end;
```

The code above also assigns the worker to the frame and adds the worker to the task group.



For a different approach to multiple workers problem see OmniThreadLibrary and databases.

7.11 OmniThreadLibrary and databases

Can you tell me how to use databases in combination with OmniThreadLibrary?

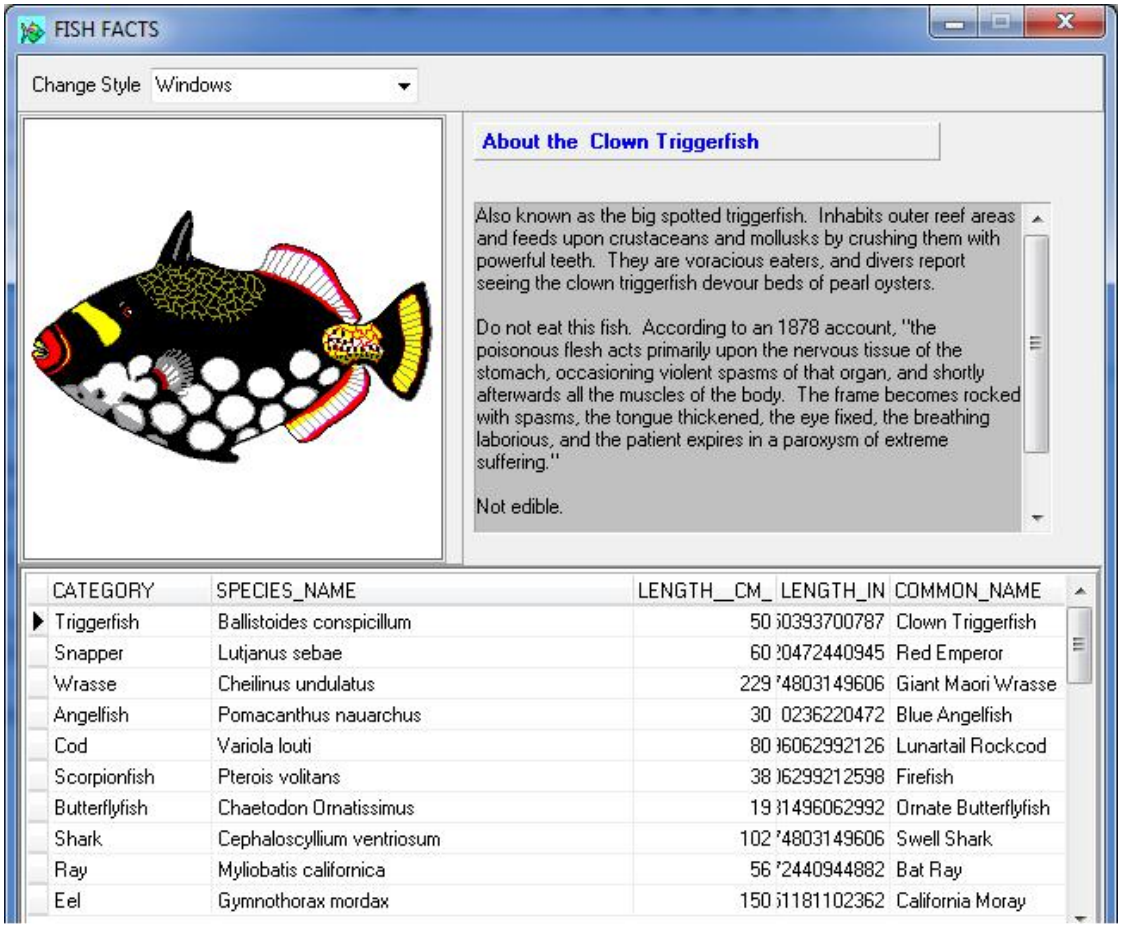
Using databases with the OmniThreadLibrary can be quite simple at times; on the other hand, it can also be quite tricky. The main problem with databases is that you have to create database components in the thread that will be using them. As the visual components (as the TDBGrid) **must** be initialized from the main thread, this implies that you can't directly connect database-aware GUI controls to database components.

Because of that you have to devise a mechanism that transfers database data from the task to the main thread (and also – if the database access is not read-only – a mechanism that will send updates to the task so that they can be applied to the database). In most cases this means that you should ignore database-aware components and just build the GUI without them. In some cases, however, you could do a lot by just splitting the existing database infrastructure at the correct point and leaving the GUI part almost unmodified.

This example explores such option.

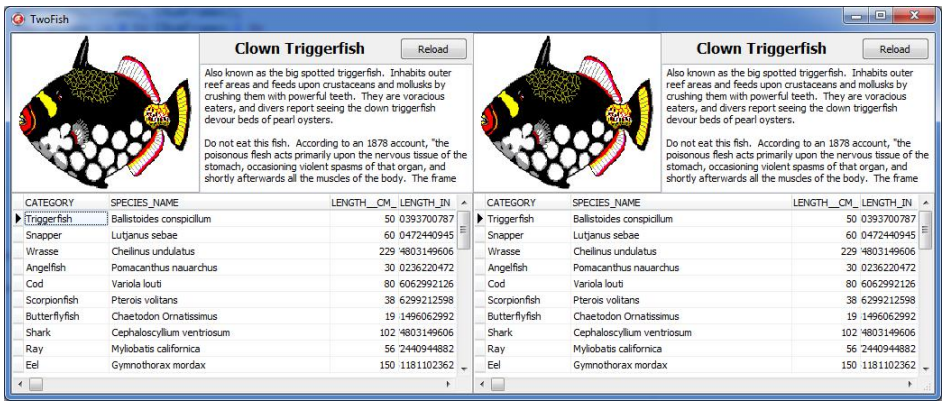
An example is included with the OmniThreadLibrary distribution in folder `examples/twofish`.

The basis for this article is the well-known Fish Facts demo program, included in Delphi's Samples[17] folder. This is a simple application that uses database-aware controls to display data from an InterBase database.



The Fish Facts demo

I have built a view-only version of Fish Facts called TwoFish which uses two frames, each containing data-aware controls and a background thread which accesses the InterBase data. Both frames are running in parallel and accessing the data at the same time.

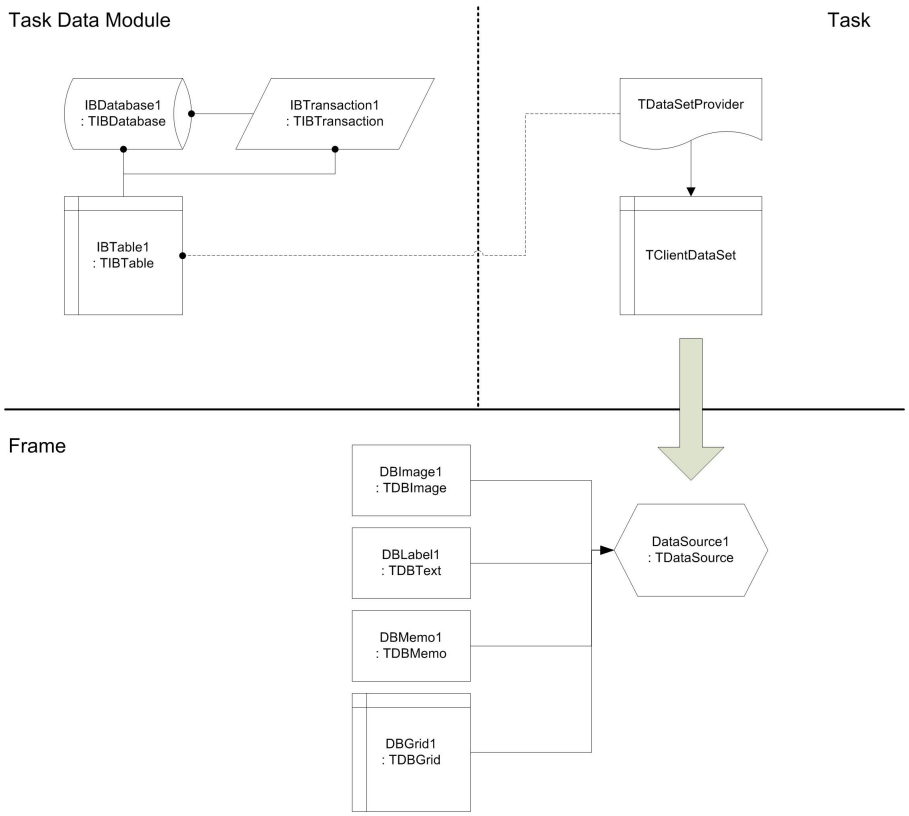


The TwoFish demo

7.11.1 Database model

To create the TwoFish, I have copied Fish Facts components `IBDatabase1`, `IBTransaction1` and `IBTable1` into a data module `twoFishDB`. This data module contains no code, only these three components. I have also set `IBDatabase1.Connected` and `IBTable1.Active` to `False`.

Then I created the frame `twoFishDB_GUI` which uses the data module `twoFishDB`. This frame contains an unconnected `TDataSource` component `DataSource1` and all data-aware components that are placed on the Fish Facts form – `TDBGrid`, `TDBImage`, `TDBText` and `TDBMemo`. They are all connected to the `DataSource1`.



TwoFish data model

Main TwoFish program creates two frames. Each frame creates a Background Worker abstraction that (inside the worker task) creates the data module and activates database and database table (more details are given below).

When data is to be retrieved, the task creates a `TClientDataSet` and a `TDataSetProvider` which 'pumps' all data from the `IBTable1` to the `TClientDataSet`. This client data set is then sent to the main form which connects it to the `DataSource1`. This automatically causes the data to be displayed in the data-aware controls. To keep the example simple, I have disabled data editing.

The most important points of this example are:

- Data module is initialized from the task, not from frame's event handlers. This way it is initialized in the thread that uses it.
- Data module is destroyed before the `OnDestroy` is called (`OnCloseQuery` is used for this purpose). If you try to destroy the data module from the `OnDestroy`, a deadlock will occur inside the Delphi RTL code.

This example shows a different approach to frame-task interaction than the Multiple workers with multiple frames – here the background worker is managed by the frame itself, not by the main form.

7.11.2 Frame and worker

The frame wraps one background task that operates on the database and contains database-aware controls displaying the database data.

The Background Worker abstraction is created in the `AfterConstruction` method and destroyed in the `BeforeDestruction` method. `AfterConstruction` creates a background worker and specifies task initializer and finalizer (`.Initialize` and `.Finalize`). Delegates provided to these two

functions (`InitializeDatabase` and `FinalizeDatabase`) are called when background worker task is created and before it is destroyed.

```
procedure TfrmTwoFishDB_GUI.AfterConstruction;
begin
  inherited;
  FWorker := Parallel.BackgroundWorker
    .Initialize(InitializeDatabase)
    .Finalize(FinalizeDatabase)
    .Execute;
end;

procedure TfrmTwoFishDB_GUI.BeforeDestruction;
begin
  CloseConnection;
  inherited;
end;

procedure TfrmTwoFishDB_GUI.CloseConnection;
begin
  if assigned(FWorker) then begin
    FWorker.Terminate(INFINITE);
    FWorker := nil;
  end;
  FreeAndNil(FDataSet);
end;
```

You may have noticed that no code was provided to execute work items. The reason behind this is that the background worker will execute different types of requests. Instead of writing `if ... then` tests to detect the work item type and trigger appropriate code, we'll pass the executor function together with each request.

`BeforeDestruction` destroys the background worker and destroys the `FDataSet` component (we'll see later why it is used).

Task initializer and finalizer are also very simple – they just create and destroy the data module. The data module is accessible to the background worker through the `taskState` variable.

```
procedure TfrmTwoFishDB_GUI.FinalizeDatabase(const taskState: TOmniValue);
begin
  FreeAndNil(FDataModule)
end;

procedure TfrmTwoFishDB_GUI.InitializeDatabase(var taskState: TOmniValue);
begin
  FDataModule := TdmTwoFishDB.Create(nil);
  taskState := FDataModule;
end;
```

7.11.2.1 Connecting to the database

Data controls are initially in an unconnected state. They are only connected when the public method `OpenConnection` is called.

```
procedure TfrmTwoFishDB_GUI.OpenConnection(const databaseName: string;
  onConnectionOpen: TNotify);
begin
  FWorker.Schedule(
    FWorker.CreateWorkItem(databaseName),
    FWorker.Config.OnExecute(ConnectToDatabase).OnRequestDone(
      procedure (const Sender: IOmniBackgroundWorker;
        const workItem: IOmniWorkItem)
      begin
        if assigned(onConnectionOpen) then
          onConnectionOpen(Self, workItem.FatalException);
```

```
12        end
13    ));
14 end;
```

OpenConnection schedules a work request that contains the database name as a parameter. It also sets the executor function (ConnectToDatabase) and an anonymous function that will be executed after the request is processed (OnRequestDone). This anonymous function returns the result of the request to the OpenConnection caller by calling the onConnectionOpen parameter. [Result in this case is exposed as an exception that is triggered if the database connection cannot be established. If the connection can be made, the workItem.FatalException function will return nil.]

The important fact to note is that the OnExecute parameter (ConnectToDatabase) is called from the worker thread and the OnRequestDone parameter (the anonymous function) is called from the thread that created the frame (the main thread).

```
1  procedure TfrmTwoFishDB_GUI.ConnectToDatabase(
2    const workItem: IOmniWorkItem);
3  var
4    dataModule: TdmTwoFishDB;
5  begin
6    dataModule := (workItem.TaskState.AsObject as TdmTwoFishDB);
7    GTwoFishLock.Acquire; //probably only necessary if using InterBase driver
8    try
9      dataModule.IBDatabase1.DatabaseName := workItem.Data.AsString;
10     dataModule.IBDatabase1.Connected := true;
11   finally GTwoFishLock.Release; end;
12 end;
```

The data module associated with the worker is accessed through the workItem.TaskState property which gives you access to the taskState variable initialized in the InitializeDatabase method. Database name is taken from the work item parameter (workItem.Data). The database name is set in the IBDatabase component and connection is established (Connected := true). If connection fails, an exception will be raised. This exception is caught by the OmniThreadLibrary and stored in the workItem object where it is later processed by the anonymous method in the OpenConnection method.

The weird Acquire/Release pair is here because of bugs in the gds32.dll – the dynamic library that handles connection to the InterBase. It turns out that gds32 handles parallel connections to the database perfectly well – as long as they are not established at the same time. In other words – you can communicate with the database on multiple connections at the same time (get data, put data, execute SQL commands ...) but you cannot establish connections in parallel. Sometimes it will work, sometimes it will fail with a mysterious access violation error in the gds32 code. That' s why the twoFishDB_GUI unit uses a global critical section to prevent multiple connections to be established at the same time. [18]

```
1 var
2   GTwoFishLock: TOmniCS;
```

7.11.2.2 Retrieving the data

To retrieve data from the database, main unit calls the Reload function. This function is also called inside the frame from the click event on the Reload button.

Reload just schedules a work request without any input. To process the request, LoadData will be called.

```
1 procedure TfrmTwoFishDB_GUI.Reload;
2 begin
```

```
  FWorker.Schedule(
    FWorker.CreateWorkItem(TOmniValue.Null),
    FWorker.Config.OnExecute(LoadData).OnRequestDone(DisplayData)
  );
end;
```

`LoadData` executes in the background worker thread. It uses a temporary `TDataSetProvider` to copy data to a freshly created `TClientDataSet`[19]. During this process, a 'Field not found'[20] exception is raised twice. If you run the program in the debugger, you'll see this exception four times (twice for each frame). You can safely ignore the exception as it is handled internally in the Delphi RTL and is not visible to the end-user.

At the end, the `TClientDataSet` that was created inside the `LoadData` is assigned to the `workItem.Result`. It will be processed (and eventually destroyed) in the main thread.

```
procedure TfrmTwoFishDB_GUI.LoadData(const workItem: IOmniWorkItem);
var
  dataModule : TdmTwoFishDB;
  resultDS   : TClientDataSet;
  tempProvider: TDataSetProvider;
begin
  dataModule := (workItem.TaskState.AsObject as TdmTwoFishDB);
  if not dataModule.IBTable1.Active then
    dataModule.IBTable1.Active := true
  else
    dataModule.IBTable1.Refresh;

  resultDS := TClientDataSet.Create(nil);

  tempProvider := TDataSetProvider.Create(nil);
  try
    tempProvider.DataSet := dataModule.IBTable1;
    resultDS.Data := tempProvider.Data;
  finally FreeAndNil(tempProvider); end;

  workItem.Result := resultDS; // receiver will take ownership
end;
```

The `DisplayData` method executes in the main thread after the request was processed (i.e., the data was retrieved). If there was an exception inside the work item processing code (`LoadData`), it is displayed. Otherwise, the `TClientDataSet` is copied from the `workItem.Result` into an internal `TfrmTwoFishDB_GUI` field and assigned to the `DataSource1.DataSet`. By doing that, all data-aware controls on the frame can access the data.

```
procedure TfrmTwoFishDB_GUI.DisplayData(
  const Sender: IOmniBackgroundWorker;
  const workItem: IOmniWorkItem);
begin
  FreeAndNil(FDataSet);

  if workItem.IsExceptional then
    ShowMessage('Failed to retrieve data. ' +
      workItem.FatalException.Message)
  else begin
    FDataSet := workItem.Result.AsObject as TClientDataSet;
    DataSource1.DataSet := FDataSet;
  end;
end;
```

7.11.3 Main program

The main program is fairly simple. In the `OnCreateEvent` two frames are created. Frame references are stored in the `FFrames` form field, declared as `array of TfrmTwoFishDB_GUI`.

```
procedure TfrmTwoFish.FormCreate(Sender: TObject);
var
  iFrame: integer;
begin
  SetLength(FFrames, CNumFrames);
  for iFrame := 0 to CNumFrames-1 do
    FFrames[iFrame] := CreateFrame(
      CFrameWidth * iFrame, 0, CFrameWidth, CFrameHeight,
      Format('Frame%d', [iFrame+1]));
  ClientWidth := CNumFrames * CFrameWidth;
  ClientHeight := CFrameHeight;
  OpenConnections;
end;
```

Next, the form is resized to twice the frame size and `OpenConnections` is called to establish database connections in all frames.

```
procedure TfrmTwoFish.OpenConnections;
var
  frame: TfrmTwoFishDB_GUI;
begin
  for frame in FFrames do
    frame.OpenConnection(CDatabaseName ,
      procedure (Sender: TObject; FatalException: Exception)
      begin
        if assigned(FatalException) then
          ShowMessage('Failed to connect to the database!')
        else
          (Sender as TfrmTwoFishDB_GUI).Reload;
      end);
end;
```

`OpenConnections` iterates over all frames and calls `OpenConnection` method in each one. Two parameters are passed to it – the database name and an anonymous method that will be executed after the connection has been established.

If connection fails, the `FatalException` field will contain the exception object raised inside the background worker's `OpenConnection` code. In such case, it will be logged. Otherwise, the connection was established successfully and `Reload` is called to load data into the frame.

Frames are destroyed from `OnCloseQuery`. It turns out that Delphi (at least XE2) will deadlock if data modules are destroyed in background threads while `OnDestroy` is running.

```
procedure TfrmTwoFish.FormCloseQuery(Sender: TObject;
  var CanClose: boolean);
var
  frame: TfrmTwoFishDB_GUI;
begin
  for frame in FFrames do
    frame.CloseConnection;
end;
```

To recapitulate, most important facts about using databases from secondary threads are:

- Always create non-visual database components in the thread that will be using them.
- Always create data-aware controls in the main thread.
- Never connect data-aware controls to non-visual database components that were created in a secondary thread.
- Wrap `TIBDatabase.Connected := true` in a critical section because of gds32 bugs.
- Destroy database tasks from `OnCloseQuery`, not from `OnDestroy` if you are using data modules in a secondary thread.
- Establish some mechanism of data passing between the database (secondary thread) and the view (main thread).

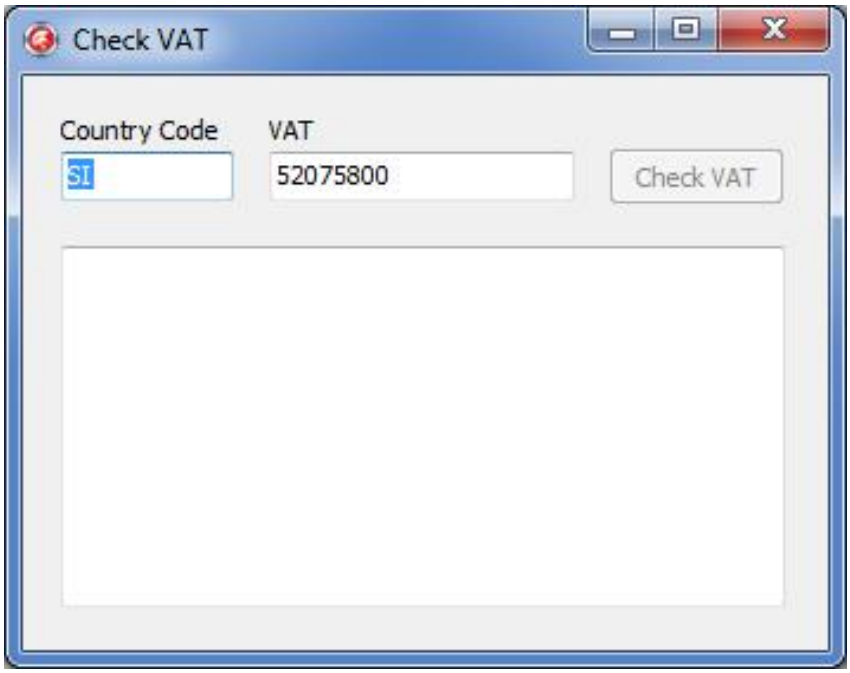
7.12 OmniThreadLibrary and COM/OLE

Can you give me any tips on how to use COM services from background threads?

It is actually very simple – you have to remember to call `CoInitializeEx` and `CoUninitialize` from the task code and then you won't have any problems.

I have put together a simple example that uses SOAP to retrieve VAT info for European companies using the SOAP service at ec.europa.eu. It is included with the OmniThreadLibrary distribution in folder `examples/checkVat`.

The program has two input fields, one for the country code (`inpCC`) and one for the VAT number (`inpVat`), a button that triggers the SOAP request (`btnCheckVat`) and a memo that displays the result (`outVatInfo`).



There's only one method – the `btnCheckVat.OnClick` handler.

```
1  procedure TfrmCheckVat.btnCheckVatClick(Sender: TObject);
2  begin
3    btnCheckVat.Enabled := false;
4    outVatInfo.Lines.Clear;
5    FRequest := Parallel.Future<checkVatResponse>(
6      function: checkVatResponse
7      var
8        request: checkVat;
9      begin
10       OleCheck(CoInitializeEx(nil, COINIT_MULTITHREADED));
11       try
12         request := checkVat.Create;
13         try
14           request.countryCode := Trim(inpCC.Text);
15           request.vatNumber := Trim(inpVat.Text);
16           Result := checkVatService.GetcheckVatPortType.checkVat(request);
17         finally FreeAndNil(request); end;
18       finally CoUninitialize; end;
19     end,
20     Parallel.TaskConfig.OnTerminated(
21       procedure (const task: IOmniTaskControl)
22       begin
23         outVatInfo.Text := FRequest.Value.name_ + #13#10 +
```

```
24            FRequest.Value.address;
25          FRequest := nil;
26          btnCheckVat.Enabled := true;
27        end
28      )
29    );
30  end;
```

This method firstlyf disables the button (so that only one request at a time can be active) and clears the output. Then it uses a Future returning a `checkVatResponse` (a type defined in the `checkVatService` unit which was generating by importing the WSDL specification). This Future will execute the SOAP request in a background task and after that the anonymous method in `Parallel.TaskConfig.OnTerminated` will be called in the main thread. This anonymous method displays the result in the `outVatInfo` control, destroys the `FRequest` Future object and enables the button.

The main Future method looks just the same as if it would be executed from the main thread except that the SOAP stuff is wrapped in `CoInitializeEx`/`CoUninitialize` calls that make sure that everything is correctly initialized for COM/OLE.

7.13 Using message queue with a TThread worker

I'm not yet using OmniThreadLibrary for multithreading, just normal `TThread`-based threads, but I would like to add OmniThreadLibrary's messaging model to my threads. How can I use OmniThreadLibrary to send and receive data to/from a thread?

The simplest way is to create two `TOmniMessageQueue` objects, one to send data to a thread and one to receive data. Alternatively, you could create a `TOmniTwoWayChannel`, which is just a simple pair of two `TOmniMessageQueue` instances. The solution below uses a former approach.

> A solution to this problem is included with the OmniThreadLibrary distribution in folder `examples/TThread communication`.

We have to handle two very similar but not completely identical parts:

- Sending data from any thread (main or background) to a `TThread` based worker.
- Sending data from any `TThread` based worker or from a form to the main thread (to a form).

Let's deal with them one by one.

7.13.1 Sending data from multiple producers to a single worker

To send data form a form to a thread, we need a message queue. This example uses a `TOmniMessageQueue` object for that purpose. An instance of this object is created in the main thread. All threads – the main thread, the worker threads, and possible other data-producing threads – use the same shared object which is written with thread-safety in mind.

7.13.1.1 Initialization and cleanup

The `TOmniMessageQueue` constructor takes a maximum queue size for a parameter. `TWorker` is just a simple `TThread` descendant which accepts the instance of the message queue as a parameter so it can read from the queue.

```
1 FCommandQueue := TOmniMessageQueue.Create(1000);
2 FWorker := TWorker.Create(FCommandQueue);
```

The shutdown sequence is fairly standard. `Stop` is used instead of `Terminate` so it can set internal event which is used to signal the thread to stop.

```
1 if assigned(FWorker) then begin
2   FWorker.Stop;
3   FWorker.WaitFor;
4   FreeAndNil(FWorker);
5 end;
6 FreeAndNil(FCommandQueue);
```

7.13.1.2 Sending data to the worker

To put some data into a queue, use its `Enqueue` method. It accepts a `TOmniMessage` record. Each `TOmniMessage` contains an integer message ID (not used in this example) and a `TOmniValue` data which, in turn, can hold any data type.

```
1 procedure TfrmTThreadComm.Query(value: integer);
2 begin
3   if not FCommandQueue.Enqueue(TOmniMessage.Create(0 {ignored}, value)) then
4     raise Exception.Create('Command queue is full!');
5 end;
```

`Enqueue` returns `False` if the queue is full. (A `TOmniMessageQueue` can only hold as much elements as specified in the constructor call.)

The example below shows how everything works correctly if two threads are started (almost) at the same time and both write to the message que

A. Units

This appendix gives a short description of OmniThreadLibrary units.

- OtlCollections

  Contains blocking collection.

- OtlComm

  Contains communication subsystem - `IOmniCommunicationEndpoint` and `TOmniTwoWayChannel`.

- OtlCommBufferTest

  This unit is used internally to test the `TOmniMessageQueue`.

- OtlCommon

  Contains common constants, types, interfaces and classes - `TOmniValue`, `TOmniRecord<T>`, `TOmniRecordWrapper`, `TOmniWaitableValue`, `TOmniValueContainer`, `TOmniCounter`, `IOmniIntegerSet`, `Environment`, `TOmniAlignedInt32`, and more.

- OtlCommon.Utils

  Contains only procedure `SetThreadName`. It is stored in a separate unit so that debugging info can be disabled for that unit.

- OtlContainerObserver

  Contains implementation of container observers.

- OtlContainers

  Contains implementation of lock-free collections - bounded stack, bounded queue, and dynamic queue.

- OtlDataManager

  Contains backend support for the ForEach abstraction.

- OtlEventMonitor

  Contains implementation of the `TOmniEventMonitor` component.

- OtlHooks

  Contains support for external hooks.

- OtlLogger

  Contains a simple log collector, used internally for testing.

- OtlParallel

  Contains implementation of high-level abstractions - Async, Async/Await, Future, Join, Parallel task, Background worker, Pipeline, Parallel for, ForEach, Fork/Join, and Map.

- OtlRegister

Contains registration code for the TOmniEventMonitor component.

- OtlSuperObject

  Contains SuperObject converters for TOmniValue.

- OtlSync

  Contains synchronization primitives - IOmniCriticalSection, TOmniCS, TOmniMREW, IOmniResourceCount, IOmniCancellationToken, Atomic<T>, Locked<T>, IOmniLockManager<K>, TWaitFor, TOmniSingleThreadUseChecker, and more.

- OtlTask

  Contains definition of the IOmniTask used in low-level multithreading.

- OtlTaskControl

  Contains definition of the IOmniTaskControl interface and full implementation of the low-level multithreading layer.

- OtlThreadPool

  Contains thread pool implementation.

B. Demo applications

OmniThreadLibrary distribution includes plenty of demo applications that will help you get started. They are stored in the `tests` subfolder. This chapter lists all tests.

- `0_Beep`: The simplest possible OmniThreadLibrary threading code.
- `1_HelloWorld`: Threaded "Hello, World" with `TOmniEventMonitor` component created in runtime.
- `2_TwoWayHello`: "Hello, World" with bidirectional communication; `TOmniEventMonitor` created in runtime.
- `3_HelloWorld_with_package`: Threaded "Hello, World" with `TOmniEventMonitor` component on the form.
- `4_TwoWayHello_with_package`: Hello, World with bidirectional communication; `TOmniEventMonitor` component on the form.
- `5_TwoWayHello_without_loop`: Hello, World with bidirectional communication, the OTL way.
- `6_TwoWayHello_with_object_worker`: Obsolete, almost totally equal to the demo 5_TwoWayHello_without_loop.
- `7_InitTest`: Demonstrates `WaitForInit`, `ExitCode`, `ExitMessage`, and `SetPriority`.
- `8_RegisterComm`: Demonstrates creation of additional communication channels.
- `9_Communications`: Simple communication subsystem tester.
- `10_Containers`: Full-blown communication subsystem tester. Used to verify correctness of the lock-free code.
- `11_ThreadPool`: Thread pool demo.
- `12_Lock`: Demonstrates `WithLock`.
- `13_Exceptions`: Demonstrates exception catching.
- `14_TerminateWhen`: Demonstrates `TerminateWhen` and `WithCounter`.
- `15_TaskGroup`: Task group demo.
- `16_ChainTo`: Demonstrates `ChainTo`.
- `17_MsgWait`: Demonstrates `MsgWait` and Windows message processing inside tasks.
- `18_StringMsgDispatch`: Calling task methods by name and address.
- `19_StringMsgBenchmark`: Benchmarks various ways of task method invocation.
- `20_QuickSort`: Parallel QuickSort demo.
- `21_Anonymous_methods`: Demonstrates the use of anonymous methods as task workers in Delphi 2009.
- `22_Termination`: Tests for `Terminate` and `Terminated`.
- `23_BackgroundFileSearch`: Demonstrates file scanning in a background thread.
- `24_ConnectionPool`: Demonstrates how to create a connection pool with OmniThreadLibrary.
- `25_WaitableComm`: Demo for `ReceiveWait` and `SendWait`.
- `26_MultiEventMonitor`: How to run multiple event monitors in parallel.

- 27_RecursiveTree: Parallel tree processing.
- 28_Hooks: Demo for the new hook system.
- 29_ImplicitEventMonitor: Demo for OnMessage and OnTerminated, named method approach.
- 30_AnonymousEventMonitor: Demo for OnMessage and OnTerminated, anonymous method approach.
- 31_WaitableObjects: Demo for the RegisterWaitObject/UnregisterWaitObject API.
- 32_Queue: Stress test for TOmniBaseQueue and TOmniQueue.
- 33_BlockingCollection: Stress test for the TOmniBlockingCollection, also demoes the use of Environment to set process affinity.
- 34_TreeScan: Parallel tree scan using TOmniBlockingCollection.
- 35_ParallelFor: Parallel tree scan using ForEach (Delphi 2009 and newer).
- 37_ParallelJoin: ParallelJoin: Join demo.
- 38_OrderedFor: Ordered ForEach loops.
- 39_Future: Futures.
- 40_Mandelbrot: Very simple parallel graphics demo.
- 41_Pipeline: Multistage parallel processes using Pipeline.
- 42_MessageQueue: Stress test for TOmniMessageQueue.
- 43_InvokeAnonymous: Demo for IOmniTask.Invoke.
- 44_Fork-Join QuickSort: QuickSort implemented using Fork/Join.
- 45_Fork-Join max: Max(array) implemented using Fork/Join.
- 46_Async: Demo for Async abstraction.
- 47_TaskConfig: Demo for task configuration with Parallel.TaskConfig.
- 48_OtlParallelExceptions: Exception handling in high-level OTL constructs.
- 49_FramedWorkers: Multiple frames each communication with own worker task.
- 50_OmniValueArray: Wrapping arrays, hashes and records in TOmniValue.
- 51_PipelineStressTest: Pipeline stress test by [Anton Alisov].
- 52_BackgroundWorker: Demo for the Background worker abstraction.
- 53_AsyncAwait: Demo for the Async/Await abstraction.
- 54_LockManager: Lock manager (IOmniLockManager<K>) demo.
- 55_ForEachProgress: Demonstrates progress bar updating from a ForEach loop.
- 56_RunInvoke: Simplified 'run & invoke' low-level API.
- 57_For: Simple and fast Parallel for.
- 58_ForVsForEach: Speed comparison between ForEach, Parallel for, and TParallel.For (XE7+).
- 59_TWaitFor: Demo for the TWaitFor class.
- 60_Map: Demonstrates the Map abstraction.
- 61_CollectionToArray: Demonstrates the TOmniBlockingCollection.ToArray method.
- 62_Console: Demonstrates how to use OmniThreadLibrary from a console application.
- 63_Service: Demonstrates how to use OmniThreadLibrary from a service application.
- 64_ProcessorGroups_NUMA: Demonstrates how to work with processor groups and NUMA nodes.
- 65_TimedTask: Demonstrates the TimedTask abstraction.
- 66_ThreadsInThreads: Demonstrates how to start OmniThreadLibrary threads from background threads.

A. Examples

OmniThreadLibrary distribution includes some complex examples, stored in the examples subfolder. This chapter lists all examples. Many are also explained in the How-to chapter.

- checkVat

  OmniThreadLibrary and COM/OLE

  Using COM/OLE from OmniThreadLibrary.

- forEach output

  Parallel for with synchronized output

  Redirecting output from a parallel ForEach loop into a structure that doesn't support multithreaded access.

- `report generator`

  Simulation of a report generator, which uses multiple Background workers to generate reports; one worker per client.

- `stringlist parser`

  Background worker and list partitioning

  Writing server-like background processing.

- `TThread communication`

  Using message queue with a TThread worker

  Using `TOmniMessageQueue` to communicate with a `TThread`-based worker.

- `twofish`

  OmniThreadLibrary and databases

  Using databases from OmniThreadLibrary.

B. Hooking into OmniThreadLibrary

The OtlHooks unit allows your code to hook into internal OmniThreadLibrary processes. Currently you can register notification methods which are called when a thread is created/destroyed, a pool is created/destroyed, or an unhandled exception 'escapes' from a task.

Exception notifications

Exception filter allows your code to be notified when an unhandled exception in a task occurs. You can also prevent exception from being stored in the IOmniTask.FatalException property.

```
type
  TExceptionFilterProc = procedure(var e: Exception; var continueProcessing: boolean);
  TExceptionFilterMeth = procedure(var e: Exception; var continueProcessing: boolean)
                         of object;

procedure RegisterExceptionFilter(filterProc: TExceptionFilterProc);overload;
procedure RegisterExceptionFilter(filterMethod: TExceptionFilterMeth);overload;
procedure UnregisterExceptionFilter(filterProc: TExceptionFilterProc);overload;
procedure UnregisterExceptionFilter(filterMethod: TExceptionFilterMeth); overload;
```

Call RegisterExceptionFilter to register a custom exception filter. Call UnregisterExceptionFilter to remove custom exception filter.

Exception filter can use application-specific logging code to log detailed information about application state. It can also free the exception object e and set it to nil, which will prevent this exception to be stored in the FatalException property.

If the filter sets continueProcessing to false, further custom exception filters won't be called. Filters are always called in the order in which they were registered.

Thread notifications

Thread notifications allow your code to be notified when a thread is created or destroyed inside the OmniThreadLibrary. This allows OmniThreadLibrary to cooperate with application-specific exception-logging code.

```
type
  TThreadNotificationType = (tntCreate, tntDestroy);
  TThreadNotificationProc = procedure(notifyType: TThreadNotificationType;
                            const threadName: string);
  TThreadNotificationMeth = procedure(notifyType: TThreadNotificationType;
                            const threadName: string) of object;

procedure RegisterThreadNotification(notifyProc: TThreadNotificationProc); overload;
procedure RegisterThreadNotification(notifyMethod: TThreadNotificationMeth); overload;
procedure UnregisterThreadNotification(notifyProc: TThreadNotificationProc); overload;
procedure UnregisterThreadNotification(notifyMethod: TThreadNotificationMeth); overload;
```

Call RegisterThreadNotification to register a thread notification method. Call UnregisterThreadNotification to unregister such method.

Notification method is always called in the context of the thread being created/destroyed.

For example, the following code fragment registers/unregisters OmniThreadLibrary threads with an application-specific thread logger.

```
procedure OtlThreadNotify(notifyType: TThreadNotificationType; const threadName: string);
var
  name: string;
begin
  case notifyType of
    tntCreate:
      begin
        if threadName <> '' then
          name := threadName
        else
          name := 'unnamed OTL thread';
        LoggerRegisterThread(name);
      end;
    tntDestroy:
      LoggerUnregisterThread;
    else
      raise Exception.Create('OtlThreadNotify: Unexpected notification type');
  end;
end;

OtlHooks.RegisterThreadNotification(OtlThreadNotify);
```

Pool notifications

Pool notifications allow your code to be notified when a thread pool is being created or destroyed. This allows the application to modifiy pool parameters on the fly.

```
type
  TPoolNotificationType = (pntCreate, pntDestroy);
  TPoolNotificationProc = procedure(notifyType: TPoolNotificationType;
                            const pool: IOmniThreadPool);
  TPoolNotificationMeth = procedure(notifyType: TPoolNotificationType;
                            const pool: IOmniThreadPool) of object;

procedure RegisterPoolNotification(notifyProc: TPoolNotificationProc); overload;
procedure RegisterPoolNotification(notifyMethod: TPoolNotificationMeth); overload;
procedure UnregisterPoolNotification(notifyProc: TPoolNotificationProc); overload;
procedure UnregisterPoolNotification(notifyMethod: TPoolNotificationMeth); overload;
```

Call `RegisterPoolNotification` to register a pool notification method. Call `UnregisterPoolNotification` to unregister such method.

You can, for example, use pool notification mechanism to set `Asy_OnUnhandledWorkerException` property whenever a thread pool is created.

```
procedure OtlPoolNotify(notifyType: TPoolNotificationType; const pool: IOmniThreadPool);
begin
  case notifyType of
    pntCreate: pool.Asy_OnUnhandledWorkerException := Asy_LogUnhandledOtlWorkerException;
    pntDestroy: pool.Asy_OnUnhandledWorkerException := nil;
    else        raise Exception.Create('OtlPoolNotify: Unexpected notification type');
  end;
end;

OtlHooks.RegisterPoolNotification(OtlPoolNotify);
```

E. ForEach internals

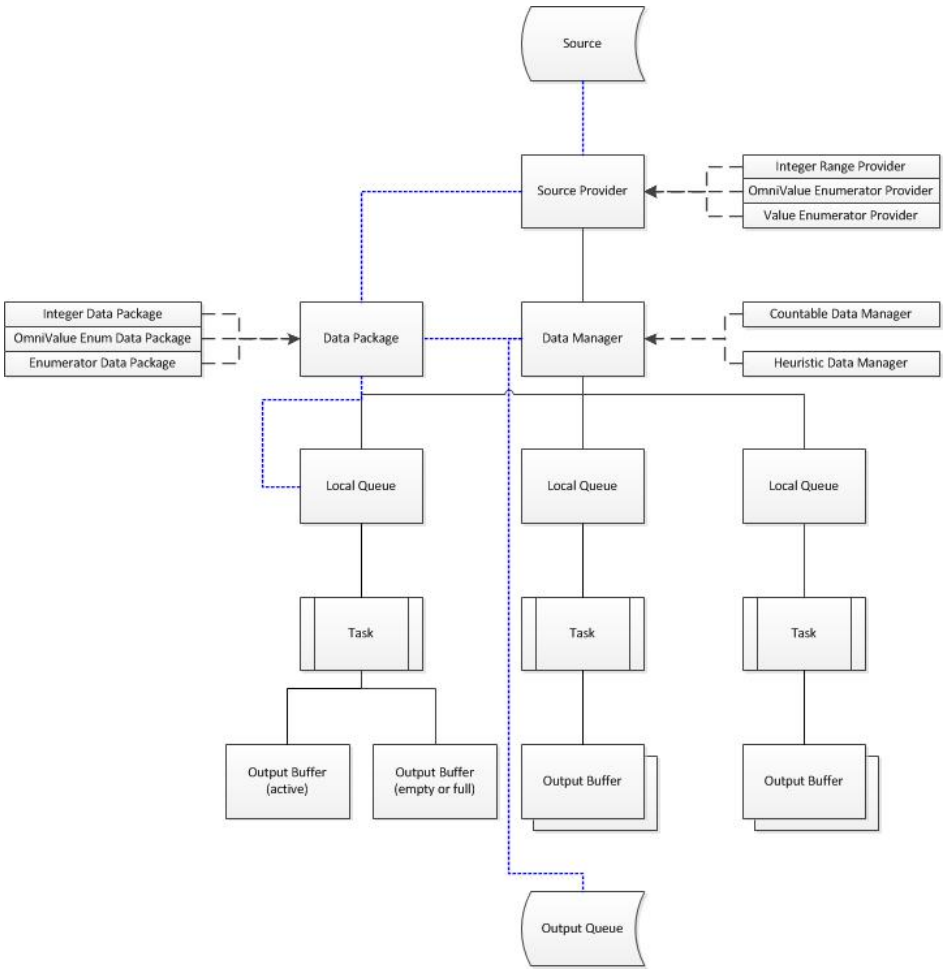
This section gives an overview of how ForEach abstraction is implemented.

Let's start with a very simple code.

```
1 Parallel.ForEach(1, 1000)
2   .Execute(
3     procedure (const elem: integer)
4     begin
5     end);
```

This simple code iterates from 1 to 1000 on all available cores in parallel and executes a simple procedure that contains no workload. All in all, the code will do nothing - but it will do it in a very complicated manner.

`ForEach` method creates new `TOmniParallelLoop<integer>` object (that's the object that will coordinate parallel tasks) and passes it a source provider - an object that knows how to access values that are being enumerated (integers from 1 to 1000 in this example).



OtlDataManager unit contains four different source providers - one for each type of source that can be passed to the `ForEach` method. If there is a need to extend `ForEach` with a new enumeration source, I would only have to add few simple methods to the OtlParallel unit and write a new source provider.

```
1 class function Parallel.ForEach(low, high: integer; step: integer):
2   IOmniParallelLoop<integer>;
3 begin
4   Result := TOmniParallelLoop<integer>.Create(
5     CreateSourceProvider(low, high, step), true);
6 end;
```

ForEach tasks are started in `InternalExecuteTask`. This method firstly creates a data manager and attaches it to the source provider (compare this with the picture above - there is one source provider and one data manager in it). Next it creates an appropriate number of tasks and calls the task-specific delegate method from each one. [This delegate wraps your parallel code and provides it with proper input (and sometimes, output). There are many calls to `InternalExecuteTask` in the OtlParallel unit, each with a different `taskDelegate` and each providing support for a different kind of the loop.]

```
procedure TOmniParallelLoopBase.InternalExecuteTask(
  taskDelegate: TOmniTaskDelegate);
var
  dmOptions    : TOmniDataManagerOptions;
  iTask        : integer;
  numTasks     : integer;
  task         : IOmniTaskControl;
  begin
    ...
    oplDataManager := CreateDataManager(oplSourceProvider,
      numTasks, dmOptions);
    ...
    for iTask := 1 to numTasks do begin
      task := CreateTask(
        procedure (const task: IOmniTask)
        begin
          ...
          taskDelegate(task);
          ...
        end,
        ...
      task.Schedule(GParallelPool);
    end;
    ...
  end;
end;
```

Data manager is a global field in the `TOmniParallelLoop<T>` object so that it can be simply reused from the task delegate. The simplest possible task delegate (below) just creates a local queue and fetches values from the local queue one by one. This results in many local queues - one per task - all connected to the same data manager.

In case you're wondering what `loopBody` is - it is the anonymous method you have passed to the `Parallel.ForEach.Execute` method.

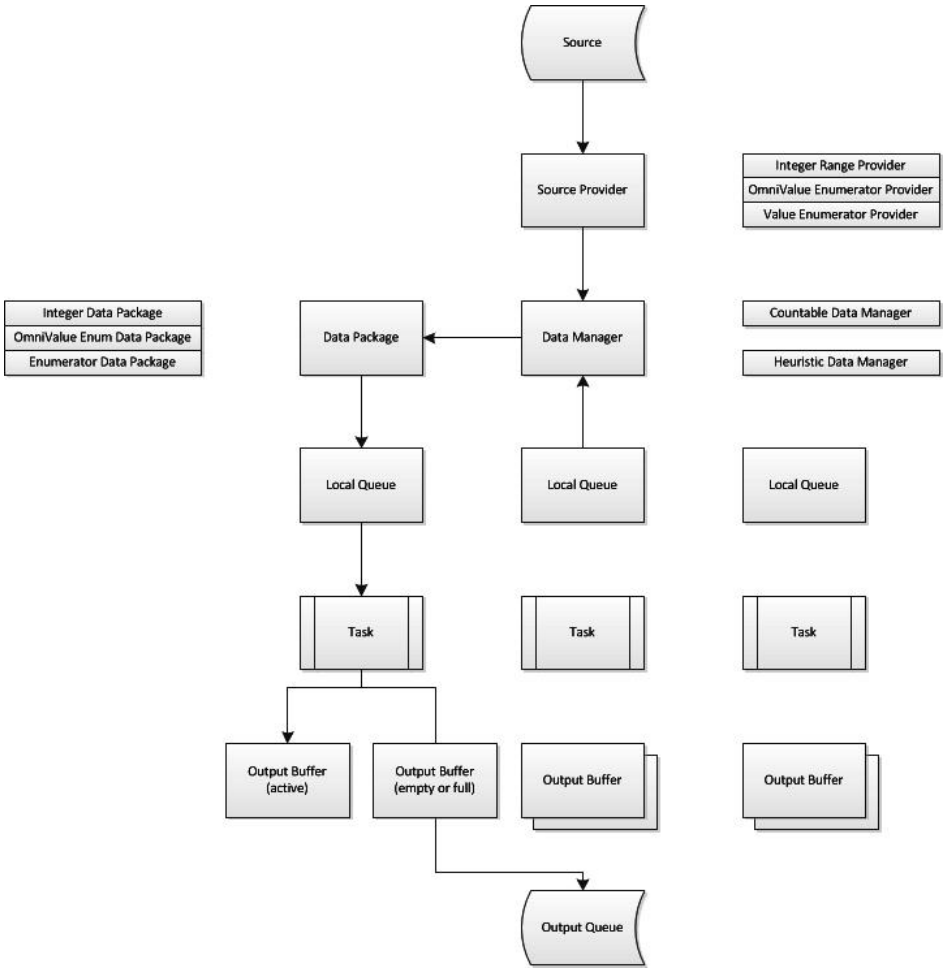
```
procedure InternalExecuteTask(const task: IOmniTask)
var
  localQueue: TOmniLocalQueue;
  value     : TOmniValue;
begin
  localQueue := oplDataManager.CreateLocalQueue;
  try
    while (not Stopped) and localQueue.GetNext(value) do
      loopBody(task, value);
  finally FreeAndNil(localQueue); end;
end;
```

Let's reiterate:

- Source provider is created.
- Data manager is created and associated with the source provider.
- Each task creates its own local queue and uses it to access the source data.
- As you'll see in the next section, local queue retrieves data in packages (data package) and sends it to an output buffer which makes sure that the output is

produced in a correct order (the output buffer part happens only if `PreserveOrder` method is called in the high-level code).

- If the task runs out of work, it requests a new data package from the data manager, which gets this data from the source provider (more on that below). If the source provider runs out of data, data manager will attempt to steal some data from other tasks.



All this was designed to provide fast data access (blocking is limited to the source provider, all other interactions are lock-free), good workload distribution (when a task runs out of work before other tasks, it will steal some work from other tasks) and output ordering (when required).

Source provider

A source provider is an object that fetches data from the enumeration source (the data that was passed to the parallel for) and repackages it into a format suitable for parallel consumption. Currently there are three source providers defined in the OtlDataManager unit.

- `TOmniIntegerRangeProvider`

  Iterates over integer ranges (just like a 'normal' `for` statement does). As such, it doesn't really fetch data from enumeration source but generates it internally.

- `TOmniValueEnumeratorProvider`

  Iterates over `IOmniValueEnumerator`, which is a special enumerator that can be accessed from multiple readers and doesn't require locking. Currently, this enumerator is only provided by the `IOmniBlockingCollection`.

- `TOmniEnumeratorProvider`

Iterates over Windows enumerators (`IEnumerator`) or Delphi enumerators (`GetEnumerator`, wrapped into `TOmniValueEnumerator` class).

All source providers descend from an abstract class `TOmniSourceProvider` which provides common source provider interface. In theory, an interface should be used for that purpose, but in practice source providers are very performance intensive and not using interfaces speeds the program by a measurable amount.

```
TOmniSourceProvider = class abstract
public
  function  Count: int64; virtual; abstract;
  function  CreateDataPackage: TOmniDataPackage; virtual; abstract;
  function  GetCapabilities: TOmniSourceProviderCapabilities;
    virtual; abstract;
  function  GetPackage(dataCount: integer;
    package: TOmniDataPackage): boolean; virtual; abstract;
  function  GetPackageSizeLimit: integer; virtual; abstract;
end;
```

Not all source providers are created equal and that's why function `GetCapabilities` returns source provider capabilities:

```
TOmniSourceProviderCapability = (
  spcCountable, // source provider that knows how much data it holds
  spcFast,      // source provider operations are O(1)
  spcDataLimit  // data package can only hold limited amount of data
);

TOmniSourceProviderCapabilities = set of
  TOmniSourceProviderCapability;
```

`TOmniIntegerRangeProvider` is both countable (it's very simple to know how many values are between 1 and 10, for example) and fast (it takes same amount of time to fetch 10 values or 10,000 values) while other two source providers are neither countable nor fast. The third capability, `spcDataLimit` is obsolete and not used. It was replaced by the `GetPackageSizeLimit` method.

The other important aspect of a source provider is the `GetPackage` method. It accesses the source (by ensuring a locked access if necessary), retrieves data and returns it in the data package. Implementation is highly dependent on the source data. For example, integer source provider just advances the current low field value and returns data package that doesn't contain bunch of values but just low and high boundaries (and that's why it is considered to be fast). Enumerator source provider locks the source, fetches the data and builds data package value by value. And in the simplest case, `TOmniValueEnumerator` source provider just fetches values and builds data package.

```
function TOmniValueEnumeratorProvider.GetPackage(dataCount: integer;
  package: TOmniDataPackage): boolean;
var
  iData     : integer;
  intPackage: TOmniValueEnumeratorDataPackage absolute package;
  timeout   : cardinal;
  value     : TOmniValue;
begin
  Assert(not StorePositions);
  Result := false;
  dataCount := intPackage.Prepare(dataCount);
  timeout := INFINITE;
  for iData := 1 to dataCount do begin
    if not vepEnumerator.TryTake(value, timeout) then
      break; //for
    intPackage.Add(value);
    timeout := 0;
    Result := true;
```

```
19    end;
20  end;
```

Data manager

Data manager is the central hub in the OtlDataManager hierarchy. It seats between multiple local queues and the single source provider and makes sure that all parallel tasks always have some work to do.

Two different data managers are implemented at the moment - a countable data manager and a heuristic data manager. The former is used if source provider is countable and the latter if it is not. Both descend from the abstract class `TOmniDataManager`.

```
1  TOmniDataManager = class abstract
2  public
3    function  CreateLocalQueue: TOmniLocalQueue; virtual; abstract;
4    function  AllocateOutputBuffer: TOmniOutputBuffer;
5      virtual; abstract;
6    function  GetNext(package: TOmniDataPackage): boolean;
7      virtual; abstract;
8    procedure ReleaseOutputBuffer(buffer: TOmniOutputBuffer);
9      virtual; abstract;
10   procedure SetOutput(const queue: IOmniBlockingCollection);
11     overload; virtual; abstract;
12 end;
```

The main difference between them lies in function `GetNextFromProvider` which reads data from the source provider (by calling its `GetPackage` method). In the countable provider this is just a simple forwarder while in the heuristic provider this function tries to find a good package size that will allow all parallel tasks to work at the full speed.

```
1  function TOmniHeuristicDataManager.GetNextFromProvider(
2    package: TOmniDataPackage; generation: integer): boolean;
3  const
4    CDataLimit = Trunc(High(integer) / CFetchTimeout_ms);
5  var
6    dataPerMs: cardinal;
7    dataSize : integer;
8    time     : int64;
9  begin
10   // the goal is to fetch as much (but not exceeding <fetch_limit>)
11   // data as possible in <fetch_timeout> milliseconds; highest amount
12   // of data is limited by the GetDataCountForGeneration method.
13   dataSize := GetDataCountForGeneration(generation);
14   if dataSize > hdmEstimatedPackageSize.Value then
15     dataSize := hdmEstimatedPackageSize.Value;
16   time := DSiTimeGetTime64;
17   Result := SourceProvider.GetPackage(dataSize, package);
18   time := DSiTimeGetTime64 - time;
19   if Result then begin
20     if time = 0 then
21       dataPerMs := CDataLimit
22     else begin
23       dataPerMs := Round(dataSize / time);
24       if dataPerMs >= CDataLimit then
25         dataPerMs := CDataLimit;
26     end;
27     // average over last four fetches for dynamic adaptation
28     hdmEstimatedPackageSize.Value := Round
29       ((hdmEstimatedPackageSize.Value / 4 * 3) +
30        (dataPerMs / 4) * CFetchTimeout_ms);
31   end;
32 end;
```

Local queue

Each parallel task reads data from a local queue, which is just a simple interface to the data manager. The most important part of a local queue is its `GetNext` method which provides the task with the next value.

```
function TOmniLocalQueueImpl.GetNext(var value: TOmniValue): boolean;
begin
  Result := lqiDataPackage.GetNext(value);
  if not Result then begin
    Result := lqiDataManager_ref.GetNext(lqiDataPackage);
    if Result then
      Result := lqiDataPackage.GetNext(value);
  end;
end;
```

Each local queue contains a local data package. `GetNext` first tries to read next value from that data package. If that fails (data packages is empty – it was already fully processed), it tries to get new data package from the data manager and (if successful) retries fetching next data from the (refreshed) data package.

`GetNext` in the data manager first tries to get next package from the source provider (via private method `GetNextFromProvider` which calls source provider's `GetPackage` method). If that fails, it tries to steal part of workload from another task.

Stealing is the feature that allows all parallel tasks to be active up to the last value being enumerated. To implement it, data manager iterates over all local queues and tries to split each local queue's data package in half. If that succeeds, half of data package is left in the original local queue and another half is returned to the local queue that requested more data.

Package splitting is highly dependent on data type. For example, integer data package just recalculates boundaries while enumerator-based packages must copy data around.

```
function TOmniValueEnumeratorDataPackage.Split(
  package: TOmniDataPackage): boolean;
var
  intPackage: TOmniValueEnumeratorDataPackage absolute package;
  iValue    : integer;
  value     : TOmniValue;
begin
  Result := false;
  for iValue := 1 to intPackage.Prepare(vedpApproxCount.Value div 2)
  do begin
    if not GetNext(value) then
      break; //for
    intPackage.Add(value);
    Result := true;
  end;
end;
```

Output ordering

Ordering (`PreserveOrder`) is usually used together with the `Into` modifier. The reason lies in the integration between the `Parallel.ForEach` infrastructure and your parallel code (the one that is executing as `Execute` payload). In the 'normal' `ForEach`, output from this parallel payload is not defined. You are allowed to generate any output in the payload but `ForEach` will know nothing about that. In this case OTL has no ability to preserver ordering because - at least from the viewpoint of the library - the parallelized code is producing no output.

When `Into` is used, however, your code uses a different signature (different parameters).

```
Parallel.ForEach(1, CMaxTest)
  .PreserveOrder
  .Into(primeQueue)
```

```
4   .Execute(
5     procedure (const value: integer; var res: TOmniValue)
6     begin
7       if IsPrime(value) then
8         res := value;
9     end);
```

Parallel payload now takes two parameters. First is – as in the more common case – the input value while the second takes the output value. As you can see from the example, the parallelized code can produce zero or one output but not more.

This small modification changes everything. As the Parallel infrastructure has control over the output parameter it can manage it internally, associate it with the input and make sure that output is generated in the same order as input was.

Let's look at the innermost code - the part that is scheduling parallel tasks. When `Into` is used, `InternalExecuteTask` executes the following quite complicated code.

```
1  InternalExecuteTask(
2    procedure (const task: IOmniTask)
3    var
4      localQueue      : TOmniLocalQueue;
5      outputBuffer_ref: TOmniOutputBuffer;
6      position        : int64;
7      result          : TOmniValue;
8      value           : TOmniValue;
9    begin
10     oplDataManager.SetOutput(oplIntoQueueIntf);
11     localQueue := oplDataManager.CreateLocalQueue;
12     try
13       outputBuffer_ref := oplDataManager.AllocateOutputBuffer;
14       try
15         localQueue.AssociateBuffer(outputBuffer_ref);
16         result := TOmniValue.Null;
17         while (not Stopped) and
18               localQueue.GetNext(position, value) do
19         begin
20           loopBody(task, value, result);
21           if not result.IsEmpty then begin
22             outputBuffer_ref.Submit(position, result);
23             result := TOmniValue.Null;
24           end;
25         end;
26       finally
27         oplDataManager.ReleaseOutputBuffer(outputBuffer_ref);
28       end;
29     finally
30       FreeAndNil(localQueue);
31     end;
32   end);
```

Important points here are:

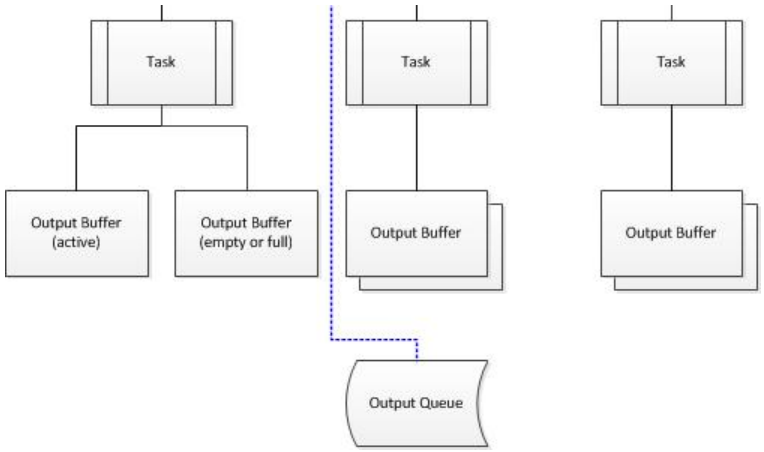
- The data manager is associated with the output queue. (The `oplIntoQueueIntf` field contains a value passed to the `Into` method.)
- A local queue is created, same as when 'normal' `ForEach` is executed.
- An output buffer is created by the data manager and associated with the local queue.
- For each input user code is executed and each non-empty output value is written into the output buffer.
- Output buffer is released, as is local queue.

The interesting part is hidden in the background; inside local queue, data manager and output buffer.

The first modification lies in the data source. When `PreserveOrder` is used, each data package knows the source position it was read from. To simplify matters, data package splitting is not used in this case. [And because of that, data stealing cannot be used causing slightly less effective use of CPU as in the simpler `ForEach` case.]

Each local queue has an output buffer set associated with it.

Each output buffer set manages two output buffers. One is active and task is writing into it and another may be either empty or full. Each output buffer is associated with an input position - just as the data package is.



When we look at data reading/writing from perspective of one task, everything is very simple. The task is reading data from a local queue (which reads data from a data package, associated with some position) and writing it to an output buffer (associated with the same position).

The tricky part comes up when the data package is exhausted (the `if not Result` branch in the code below).

```
function TOmniLocalQueueImpl.GetNext(var position: int64; var value: TOmniValue): boolean;
begin
  Result := lqiDataPackage.GetNext(position, value);
  if not Result then begin
    lqiBufferSet.ActiveBuffer.MarkFull;
    lqiBufferSet.ActivateBuffer;
      // this will block if alternate buffer is also full
    Result := lqiDataManager_ref.GetNext(lqiDataPackage);
    if Result then begin
      Result := lqiDataPackage.GetNext(position, value);
      if Result then
        lqiBufferSet.ActiveBuffer.Range := lqiDataPackage.Range;
    end;
  end;
end;
```

First, the currently active buffer is marked as full. This causes `NotifyBufferFull` to be called (see below). Then, alternate buffer is activated. This call (`ActivateBuffer`) will actually block if alternate buffer is not free. In this case, the current thread is blocked until one of its buffers is written into the output queue.

From this point on, `GetNext` proceeds in the same way as when used in the simple `ForEach`, except that it sets active buffer's position whenever new data package is read from the data manager.

The other part of the magic happens in the method that is called from `MarkFull`. It walks the buffer list and checks if there are any output buffers that are a) full and b) destined for the

current output position. Such buffers are copied to the output and returned into use.

```
procedure TOmniBaseDataManager.NotifyBufferFull(
  buffer: TOmniOutputBufferImpl);
begin
  // Remove buffer from the list. Check if next buffer is waiting in
  // the list. Copy buffer if it is full and repeat the process.
  dmBufferRangeLock.Acquire;
  try
    while (dmBufferRangeList.Count > 0) and
          (BufferList[0].Range.First = dmNextPosition) and
          BufferList[0].IsFull do
    begin
      buffer := TOmniOutputBufferImpl(
        dmBufferRangeList.ExtractObject(0));
      dmNextPosition := buffer.Range.Last + 1;
      buffer.CopyToOutput;
    end;
  finally dmBufferRangeLock.Release; end;
end;
```

To recap:

- Each data buffer is associated with a position.
- Each local queue has two output buffers, one is active and another is either free or full.
- Each output buffer is also associated with a position.
- Local queue writes data to an output buffer.
- When a buffer is full, it is put into a list of waiting buffers. At that moment all appropriate waiting buffers are copied to output.

F. Hyperlinks

About me
http://primoz.gabrijelcic.org

Assigning Data Directly (to TClientDataSet)
http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/devwin32/fhxr18643_xml.html

Async/Await in Delphi
http://www.thedelphigeek.com/2012/07/asyncawait-in-delphi.html

Async/Await in .NET
https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/

AsyncCalls
http://andy.jgknet.de/blog/bugfix-units/asynccalls-29-asynchronous-function-calls/

Blaise Pascal Magazine
https://www.blaisepascal.eu/

Blocking collection in .NET
https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/blockingcollection-overview

Busy-Wait Initialization
http://www.thedelphigeek.com/2011/12/busy-wait-initialization.html

Cover page © Dave Gingrich\ https://www.flickr.com/photos/ndanger/2744507570/

Critical section
https://en.wikipedia.org/wiki/Critical_section

Delphi - package versions
http://docwiki.embarcadero.com/RADStudio/en/Compiler_Versions

Delphi High Performance
https://www.packtpub.com/application-development/delphi-high-performance

Delphi XE2 Foundations
https://delphihaven.wordpress.com

Delphi XE2 Foundations, Part 3 (Amazon)
https://www.amazon.com/product-reviews/B008BO0TFI

Delphinus Websetup
http://memnarch.bplaced.net/blog/pojects/all-downloads/?did=19

EU VAT validation service WDSL
http://ec.europa.eu/taxation_customs/vies/checkVatService.wsdl

Event (synchronization primitive)
https://en.wikipedia.org/wiki/Event_(synchronization_primitive)

FastMM memory manager
https://github.com/pleriche/FastMM4

Fixing TCriticalSection
https://www.delphitools.info/2011/11/30/fixing-tcriticalsection/

Fluent interface
https://en.wikipedia.org/wiki/Fluent_interface

GetNumaProximityNodeEx
https://msdn.microsoft.com/en-us/library/windows/desktop/dd405495(v=vs.85).aspx

GpDelphiUnits – repository
https://github.com/gabr42/GpDelphiUnits

How --await-- works
https://blogs.remobjects.com/2012/08/08/how-await-works/

Installing Delphinus
https://github.com/Memnarch/Delphinus/wiki/Installing-Delphinus

Intentional programming
https://en.wikipedia.org/wiki/Intentional_programming

ITask
http://docwiki.embarcadero.com/Libraries/en/System.Threading.ITask

Junk Generator Speed Problem
https://stackoverflow.com/questions/7292741/junk-generator-speed-problem

Monitor Magazine
http://www.monitor.si/

MsgWaitForMultipleObjects
https://msdn.microsoft.com/en-us/library/windows/desktop/ms684242(v=vs.85).aspx

MsgWaitForMultipleObjectsEx
https://msdn.microsoft.com/en-us/library/windows/desktop/ms684245(v=vs.85).aspx

Multithreading – The Delphi Way
http://www.nickhodges.com/MultiThreadingInDelphi/ToC.html

NUMA Support
https://msdn.microsoft.com/en-us/library/windows/desktop/aa363804(v=vs.85).aspx

Observer pattern
https://en.wikipedia.org/wiki/Observer_pattern

OmniThreadLibrary
http://www.omnithreadlibrary.com

OmniThreadLibrary – articles on The Delphi Geek blog
http://www.thedelphigeek.com/search/label/OmniThreadLibrary

OmniThreadLibrary – download
http://www.omnithreadlibrary.com/omnithreadlibrary/download.htm

OmniThreadLibrary – Google+ Community

https://plus.google.com/communities/112307748950248514961

OmniThreadLibrary – questions on StackOverflow
https://stackoverflow.com/search?q=omnithreadlibrary

OmniThreadLibrary – repository
https://github.com/gabr42/OmniThreadLibrary

OmniThreadLibrary – webinars
https://gumroad.com/thedelphigeek

Packt Publishing
https://www.packtpub.com/

Processor Groups
https://msdn.microsoft.com/en-us/library/windows/desktop/dd405503(v=vs.85).aspx

Queue (abstract data type)
https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

Quicksort
https://en.wikipedia.org/wiki/Quicksort

RegisterWaitForSingleObject
https://msdn.microsoft.com/en-us/library/windows/desktop/ms685061(v=vs.85).aspx

Speed comparison: Variant, TValue, and TOmniValue
http://www.thedelphigeek.com/2010/03/speed-comparison-variant-tvalue-and.html

Stack (abstract data type)
https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

TCriticalSection
http://docwiki.embarcadero.com/Libraries/en/System.SyncObjs.TCriticalSection

The Deadlock Empire
http://deadlockempire.4delphi.com/delphi/

The Delphi Magazine – reprints of my articles
http://www.thedelphigeek.com/search/label/The%20Delphi%20Magazine

The Little Book of Semaphores
http://greenteapress.com/wp/semaphores/

Threads
https://en.wikipedia.org/wiki/Thread_(computing)

ThreadSafe Lock Manager: [1] Design
http://www.thedelphigeek.com/2013/04/threadsafe-lock-manager-1-design.html

ThreadSafe Lock Manager: [2] Code
http://www.thedelphigeek.com/2013/04/threadsafe-lock-manager-2-code.html

ThreadSafe Lock Manager: [3] Test
http://www.thedelphigeek.com/2013/04/threadsafe-lock-manager-3-test.html

TMonitor
http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.TMonitor

TMonitor.Enter
http://docwiki.embarcadero.com/Libraries/en/System.TMonitor.Enter

TMonitor.Exit
http://docwiki.embarcadero.com/Libraries/en/System.TMonitor.Exit

TMultiReadExclusiveWriteSynchronizer
http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.SysUtils.TMultiReadExclusiveWriteSynchronizer

TParallel.For
http://docwiki.embarcadero.com/Libraries/en/System.Threading.TParallel.For

TSemaphore
http://greenteapress.com/wp/semaphores/

TThread
http://docwiki.embarcadero.com/Libraries/en/System.Classes.TThread

TValue
http://docwiki.embarcadero.com/Libraries/en/System.Rtti.TValue

Variant
http://docwiki.embarcadero.com/Libraries/en/System.Variant

WaitForMultipleObjects
https://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx

Working with TMultiReadExclusiveWriteSynchronizer
http://edn.embarcadero.com/article/28258