
Parallel Programming
with
OmniThreadLibrary
Primož Gabrijelčič
thedelphigeek.org

Parallel Programming with OmniThreadLibrary

Primož Gabrijelčič

This book is for sale at http://leanpub.com/omnithreadlibrary

This version was published on 2019-03-01

Tweet This Book!

Please help Primož Gabrijelčič by spreading the word about this book on Twitter!

The suggested hashtag for this book is #omnithreadlibrary.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#omnithreadlibrary

Contents

Sample Book

This is just the sample of the real book. It includes selected parts from some chapters.

You can buy the book on the LeanPub[1].

[1] http://leanpub.com/omnithreadlibrary

About me

I started programming on 8-bit micros in 1980s which gave me a never fully satisfied need to push as much as possible from available hardware. That inheritance brought me quite organically to the current job where I program high-availability server applications used in the broadcasting industry.

As a side result of this focus my open source projects also tend to deal with performance. I wrote a source-instrumenting profiler for Delphi, GpProfile, which is now obsolete and no longer used. Another project, a parallel programming library for Delphi called OmniThreadLibrary quickly became a favourite in the Delphi community and is now used by many thousand users all over the world.

I enjoy teaching almost as much as coding. In my younger years I wrote for leading Slovenian technical magazine 'Monitor'[2], for topical 'The Delphi Magazine' and later for 'Blaise Pascal Magazine'[3]. Most of my articles written for 'The Delphi Magazine' are reprinted with permission on my blog 'The Delphi Geek'[4].

Lately my focus has shifted from writing for magazines to presenting on Delphi conferences, organizing workshops (in real life or online), consulting and writing books. Besides the book you are reading, I also wrote 'Delphi High Performance'[5] and 'Hands-On Design Patterns with Delphi'[6], both published by 'Packt Publishing'[7].

Find more about me at http://primoz.gabrijelcic.org.

[2] http://www.monitor.si/
[3] https://www.blaisepascal.eu/
[4] http://www.thedelphigeek.com/search/label/The%20Delphi%20Magazine
[5] https://www.packtpub.com/application-development/delphi-high-performance
[6] https://www.packtpub.com/application-development/hands-design-patterns-delphi
[7] https://www.packtpub.com/

Credits

I would like to thank Gorazd Jernejc (also known as GJ) for his contribution to the OmniThreadLibrary project. Gorazd, OTL wouldn't be the same without you!

OmniThreadLibrary would not be the same without the contributors (listed in alphabetical order): ajasja, Alex Egorov, Alexander Alexeev, andi, Anton Alisov, arioch, Dean Hill, dottor_jeckill, geskill, Hallvard Vassbotn, HHasenack, Jamie, Lee Nover, M.J. Brandt, Mason Wheeler, Mayjest, meishier, morlic, Passella, Qmodem, ring.nic, Steve Maughan, Unspoken, Uwe Raabe, VyPu, Zarko.

Great thanks go to Pierre le Riche who wrote beautiful FastMM[8] memory manager and allowed me to include his fine work in the distribution.

I would like to thank Joe C. Hecht for reading my book from start from finish and providing me with many suggestions that helped make this book even better.

Great thanks go also to the platform which allows me to self-publish my books - LeanPub[9]. This book would never happen without the hard-working people running the LeanPub.

The ProWritingAid[10] tool was indispensable in fixing style and grammar errors.

Cover page was photographed by © Dave Gingrich[11].

[8] https://github.com/pleriche/FastMM4
[9] https://leanpub.com/
[10] https://prowritingaid.com/
[11] https://www.flickr.com/photos/ndanger/2744507570/

Introduction

This is a book about OmniThreadLibrary[12], a multi-threading library for the Embarcadero Delphi rapid development environment.

To follow the book, the reader should have some understanding about the multi-threading programming. If you are new to multi-threading, I recommend reading the 'Multithreading - The Delphi Way'[13] by Martin Harvey. That book is an oldie, but goldie.

A more up-to-date overview of Delphi multi-threading capabilities was published in the 'Delphi XE2 Foundations, Part 3'[14] by Chris Rolliston, available on Amazon[15].

Formatting conventions

This book covers the latest official OmniThreadLibrary release – 3.07.7.

When a part of the book covers a different version, a [version tag] in superscript will show relevant version or versions. Version numbers (f.i. `2.1`) are used for older releases and SVN revision numbers (f.i. `r1184`) are used for a functionality that was added after the last official release.

A single version or revision number (f.i. [r1184]) shows that the topic in question was introduced in this version and that it is still supported in the current release.

A range of two versions (f.i. [1.0-1.1]) shows that the topic was introduced in the first version (`1.0` in this example) and that it was supported up to the second version (`1.1`). After that, support for this topic was removed, or it was changed so much that an additional section was added to describe the new functionality.

Learn more

A good way to learn more about the OmniThreadLibrary is to go through the included demos. They are part of the standard OmniThreadLibrary distribution and you should have them on the disk already if you have installed OmniThreadLibrary. Each demo deals with a very limited subset of OmniThreadLibrary functionality so they are fairly easy to understand.

From time to time I present OmniThreadLibrary in live webinars. Recordings are available on Gumroad[16].

I frequently post about the OmniThreadLibrary on my blog where the articles relevant to OmniThreadLibrary are specifically tagged with the 'OmniThreadLibrary' tag[17].

[12] http://www.omnithreadlibrary.com
[13] http://www.nickhodges.com/MultiThreadingInDelphi/ToC.html
[14] https://delphihaven.wordpress.com
[15] https://www.amazon.com/product-reviews/B008BO0TFI
[16] https://gumroad.com/thedelphigeek
[17] http://www.thedelphigeek.com/search/label/OmniThreadLibrary

Support is also available on StackOverflow[18] (tag the question with 'omnithreadlibrary') and on the OmniThreadLibrary forum[19].

[18] http://www.stackoverflow.com
[19] https://en.delphipraxis.net/forum/32-omnithreadlibrary/

Release notes

2018-02-28

- Version 1.0 of the book released. It covers OmniThreadLibrary version 3.07.5.

2019-03-01

- Version 1.01 of the book released. It covers OmniThreadLibrary version 3.07.7.
- Numerous style and grammar errors were fixed.

1. Introduction to OmniThreadLibrary

OmniThreadLibrary[1] is a multi-threading library for Delphi, written mostly by the author of this book (see Credits for a 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`[2] or with AsyncCalls[3]). 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`[4] 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 multi-threaded parts, takes care of synchronisation and handles other menial tasks.

1.1 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 framework aren't supported in Delphi 2009 and 2010 due to compiler bugs.

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

1.2 License

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

[1] http://www.omnithreadlibrary.com
[2] http://docwiki.embarcadero.com/Libraries/en/System.Classes.TThread
[3] http://andy.jgknet.de/blog/?page_id=100
[4] http://docwiki.embarcadero.com/Libraries/en/System.Threading.ITask

In simpler words, the license says:

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 website.
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 a support contract for the OmniThreadLibrary, please contact me.

1.3 Installation

1. Download the last stable edition (download link is available at the OmniThreadLibrary site[5]), or download the latest state from the repository[6]. Typically, it is safe to follow the repository trunk as only tested code is committed.

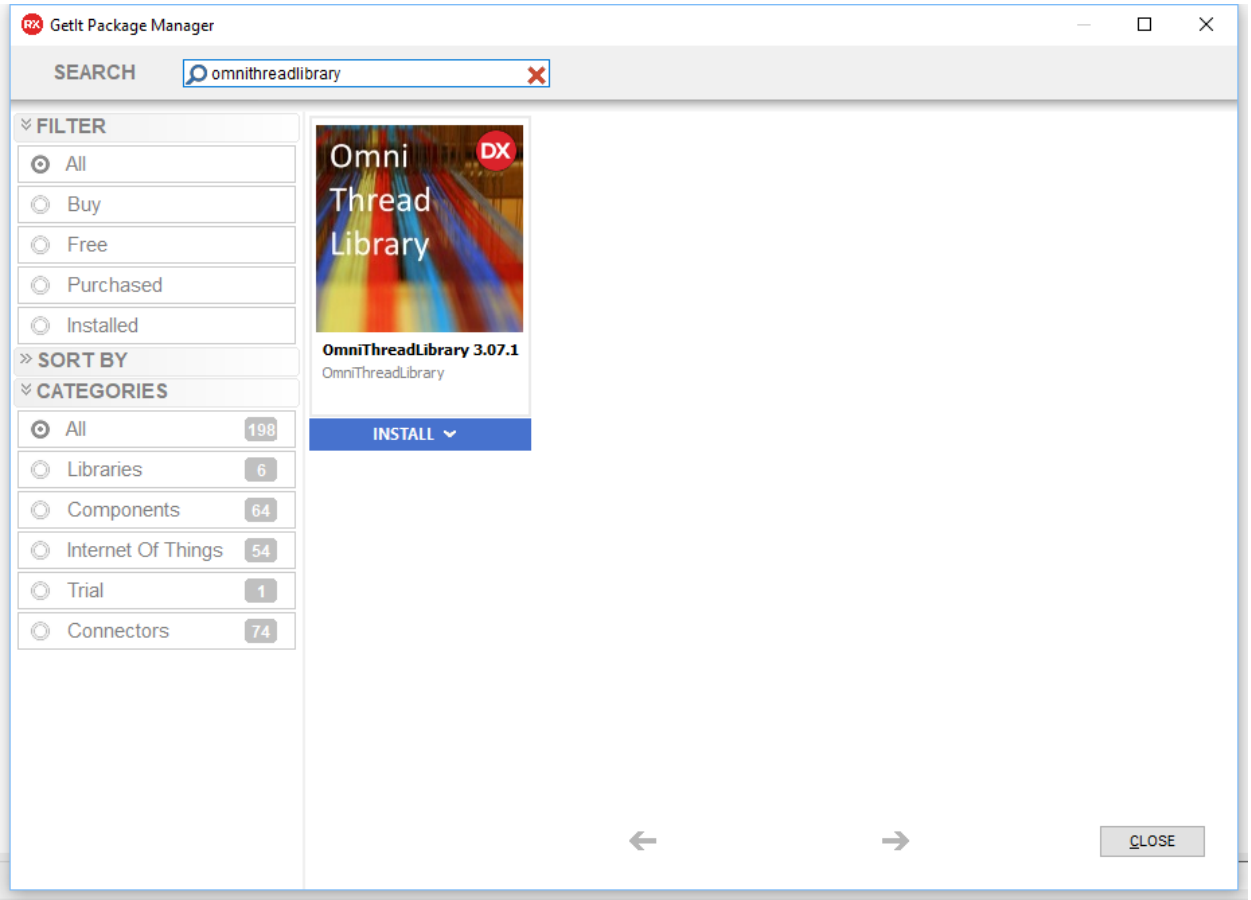
[5] http://www.omnithreadlibrary.com/omnithreadlibrary/download.htm
[6] https://github.com/gabr42/OmniThreadLibrary

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[7] 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!

1.3.1 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.

[7] https://github.com/gabr42/GpDelphiUnits

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

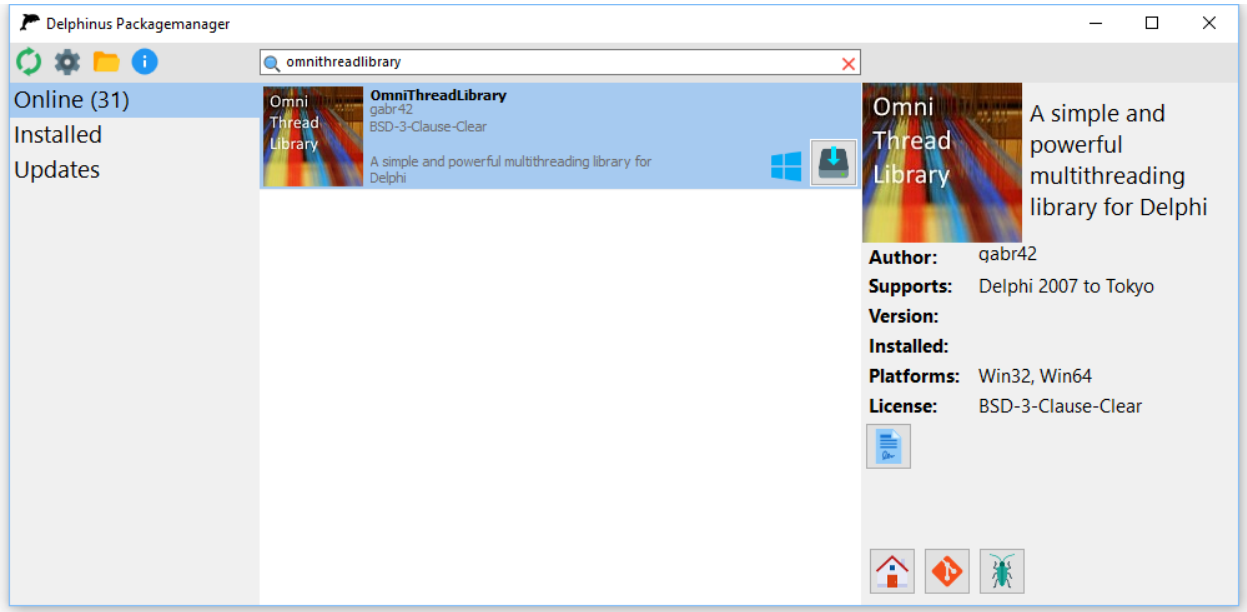
1.3.2 Installing with Delphinus

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[8].
    a. 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[9].

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.

[8] http://memnarch.bplaced.net/blog/pojects/all-downloads/?did=19
[9] https://github.com/Memnarch/Delphinus/wiki/Installing-Delphinus

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).

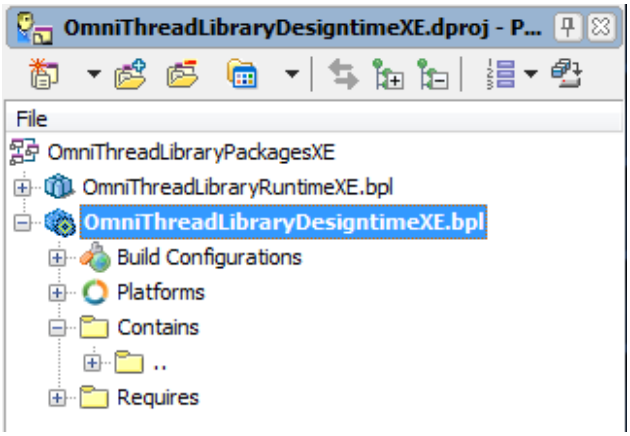
1.3.3 Installing design package

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 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.

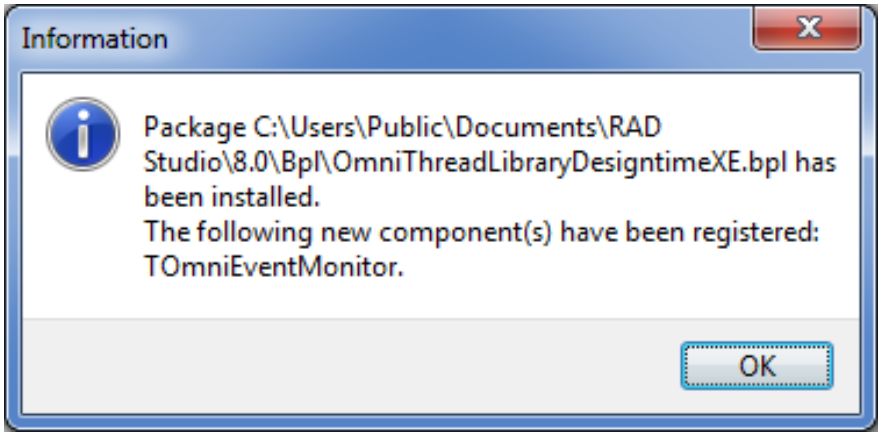
To compile and install the package containing this component, follow these steps:

- 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**[10] 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.

[10] http://docwiki.embarcadero.com/RADStudio/en/Compiler_Versions



- 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.

1.4 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 multi-threading.

OmniThreadLibrary was designed to become a "VCL for multi-threading" – a library that will make typical multi-threading tasks really simple but still allow you to dig deeper and mess with the multi-threading 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.

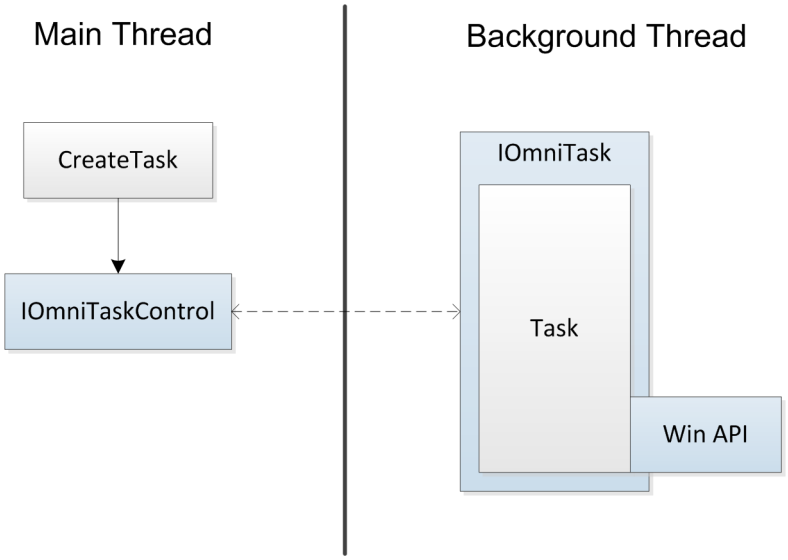
By moving most of the critical multi-threaded code into reusable components (classes and high-level abstractions), OmniThreadLibrary allows you to write better multithreaded code faster.

1.5 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 a 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 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.

1.6 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* multi-threaded 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[11] game.

[11] http://deadlockempire.4delphi.com/delphi/

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.

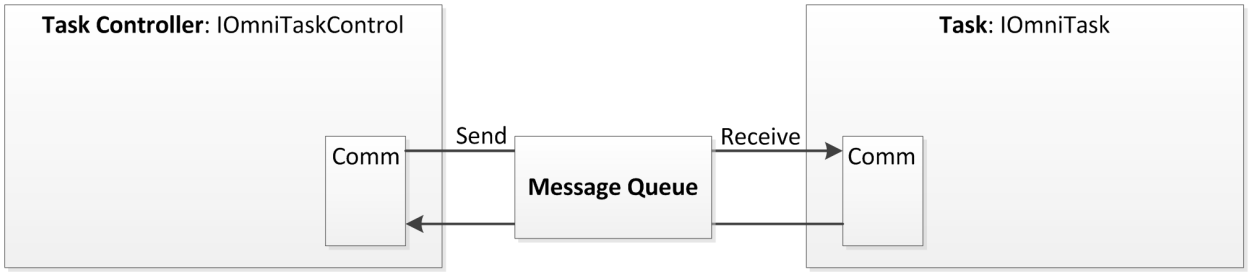
OmniThreadLibrary uses custom lock-free structures to transfer data between the task and its owner (or directly between two tasks). The system is tuned for high data rates and can transfer more than million messages per second. However, in some situations shared data approach is necessary, and that's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



1.7 Message loop required

Because of 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 thread from a console application requires more work. Additional work is also required if you are creating OmniThreadLibrary threads from background threads.

1.7.1 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 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.

1.7.2 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 an OTL task** button creates a task that will create a *Future*.

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

The important part of this demo 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.

```
function TWorker.Initialize: boolean;
begin
  Result := inherited Initialize;
  if Result then begin
    Task.Comm.Send(WM_LOG, Format('[%d] Starting a Future', [GetCurrentThreadID]));
    FCalc := Parallel.Future<integer>(Asy_DoTheCalculation,
      Parallel.TaskConfig
        .OnMessage(MSG_STATUS,
          procedure(const workerTask: IOmniTaskControl; const msg: TOmniMessage)
          begin
            // workerTask = task controller for Parallel.Future worker thread
            // Task = TWorker.Task = interface of the TWorker task
            Task.Comm.Send(WM_LOG, Format('[%d] Future sent a message: %s',
              [GetCurrentThreadID, msg.MsgData.AsString]));
          end)
        .OnTerminated(
          procedure
          begin
            Task.Comm.Send(WM_LOG, Format('[%d] Future terminated, result = %d',
              [GetCurrentThreadID, FCalc.Value]));
            FCalc := nil;
            Task.Comm.Send(WM_LOG, Format('[%d] Terminating worker',
              [GetCurrentThreadID]));
            // Terminate TWorker
            Task.Terminate;
          end));
  end;
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`).

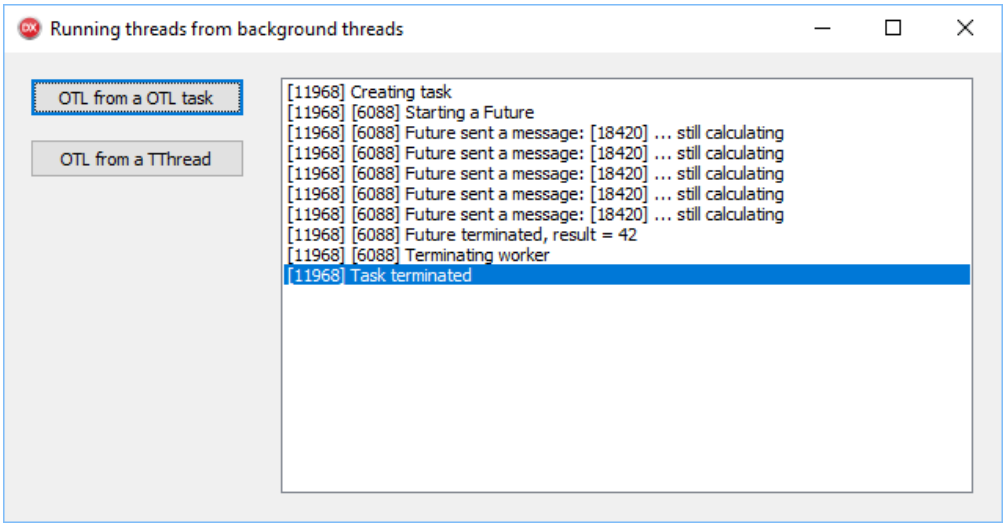
`OnMessage` takes a message that was sent from the future, adds some text and current thread ID and sends new message to the form where it is logged in the `WMLog` method (not shown here).

`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 simply 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).



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.

1.7.3 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;
  Log('Terminating worker');
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`).

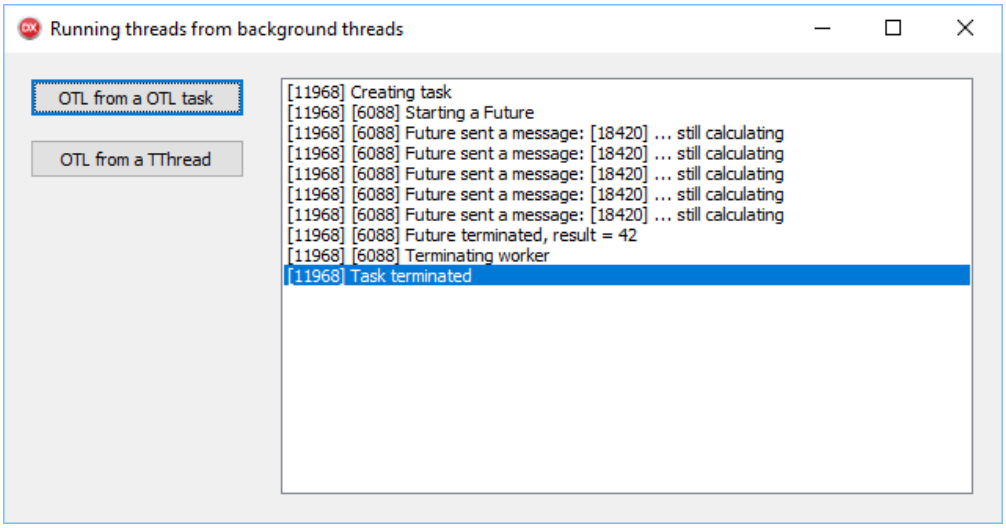
At the end it cleans up the future and exits. That destroys the `TWorkerThread` thread.

Calling `MsgWaitForMultipleObjects`[12] 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`.

[12] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684242(v=vs.85).aspx

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



1.8 TOmniValue

A `TOmniValue` (part of the *OtlCommon* unit) is a data type 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 implemented as a smart record (a record with functions and operators) which functions similarly to a `Variant`[13] or `TValue`[14] but is faster[15]. 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 a 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.

[13] http://docwiki.embarcadero.com/Libraries/en/System.Variant
[14] http://docwiki.embarcadero.com/Libraries/en/System.Rtti.TValue
[15] http://www.thedelphigeek.com/2010/03/speed-comparison-variant-tvalue-and.html

1.8.1 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, the getter for the `AsString` property is called `CastToString`. Internally it calls `TryCastToString`, which is 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);
```

```
13      ovtInteger:    value := IntToStr(ovData);
14      ovtDouble,
15      ovtDateTime,
16      ovtExtended:   value := FloatToStr(AsExtended);
17      ovtAnsiString: value := string((ovIntf as IOmniAnsiStringData).Value);
18      ovtString:     value := (ovIntf as IOmniStringData).Value;
19      ovtWideString: value := (ovIntf as IOmniWideStringData).Value;
20      ovtVariant:    value := string(AsVariant);
21      else Result := false;
22    end;
23  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.

```
1   function  TryCastToAnsiString(var value: AnsiString): boolean;
2   function  TryCastToBoolean(var value: boolean): boolean;
3   function  TryCastToCardinal(var value: cardinal): boolean;
4   function  TryCastToDouble(var value: Double): boolean;
5   function  TryCastToDateTime(var value: TDateTime): boolean;
6   function  TryCastToException(var value: Exception): boolean;
7   function  TryCastToExtended(var value: Extended): boolean;
8   function  TryCastToInt64(var value: int64): boolean;
9   function  TryCastToInteger(var value: integer): boolean;
10  function  TryCastToInterface(var value: IInterface): boolean;
11  function  TryCastToObject(var value: TObject): boolean;
12  function  TryCastToPointer(var value: pointer): boolean;
13  function  TryCastToString(var value: string): boolean;
14  function  TryCastToVariant(var value: Variant): boolean;
15  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.

```
1   function  CastToAnsiStringDef(const defValue: AnsiString): AnsiString;
2   function  CastToBooleanDef(defValue: boolean): boolean;
3   function  CastToCardinalDef(defValue: cardinal): cardinal;
4   function  CastToDoubleDef(defValue: Double): Double;
5   function  CastToDateTimeDef(defValue: TDateTime): TDateTime;
6   function  CastToExceptionDef(defValue: Exception): Exception;
7   function  CastToExtendedDef(defValue: Extended): Extended;
8   function  CastToInt64Def(defValue: int64): int64;
9   function  CastToIntegerDef(defValue: integer): integer;
10  function  CastToInterfaceDef(const defValue: IInterface): IInterface;
11  function  CastToObjectDef(defValue: TObject): TObject;
12  function  CastToPointerDef(defValue: pointer): pointer;
```

```
13  function  CastToStringDef(const defValue: string): string;
14  function  CastToVariantDef(defValue: Variant): Variant;
15  function  CastToWideStringDef(defValue: WideString): WideString;
```

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

```
1  function TOmniValue.CastToObjectDef(defValue: TObject): TObject;
2  begin
3    if not TryCastToObject(Result) then
4      Result := defValue;
5  end;
```

Function `LogValue` [3.07.6] returns a string containing both the type of the stored data and stored value.

```
1  function  LogValue: string;
```

This function is useful for data logging and debugging. See the source code for details.

1.8.2 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.

```
1  function  IsAnsiString: boolean;
2  function  IsArray: boolean;
3  function  IsBoolean: boolean;
4  function  IsEmpty: boolean;
5  function  IsException: boolean;
6  function  IsFloating: boolean;
7  function  IsDateTime: boolean;
8  function  IsInteger: boolean;
9  function  IsInterface: boolean;
10 function  IsInterfacedType: boolean;
11 function  IsObject: boolean;
12 function  IsOwnedObject: boolean;
13 function  IsPointer: boolean;
14 function  IsRecord: boolean;
15 function  IsString: boolean;
16 function  IsVariant: boolean;
17 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;
```

1.8.3 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.

1.8.4 Operators

`TOmniValue` also implements several `Implicit` operators which help with automatic conversion to and from different data types. Internally, they are implemented as an 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;
```

```
14 class operator Implicit(const a: TOmniValue): AnsiString;
15 class operator Implicit(const a: TOmniValue): int64;
16 class operator Implicit(const a: TOmniValue): TObject;
17 class operator Implicit(const a: TOmniValue): Double;
18 class operator Implicit(const a: TOmniValue): Exception;
19 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.

1.8.5 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 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>`. Since [3.07.7] `CastTo<T>` supports casting into an interface on Delphi XE3 and newer.

`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.

1.8.6 Array access

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

```
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>;
```

1.8.7 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;

ov := TOmniValue.FromRecord<TTimeStamp>(ts);
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.

```
class function TOmniValue.FromRecord<T>(const value: T): TOmniValue;
begin
  Result.SetAsRecord(
    CreateAutoDestroyObject(
      TOmniRecordWrapper<T>.Create(value)));
end;
```

1.8.8 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`.

```
property  AsOwnedObject: TObject;
function  IsOwnedObject: boolean;
property OwnsObject: boolean;
```

When an object-owning `TOmniValue` goes out of scope, it automatically destroys the owned object.

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

1.8.9 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.

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

1.8.10 Low-level methods

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

```
procedure _AddRef;
procedure _Release;
procedure _ReleaseAndClear;
function  RawData: PInt64;
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 a pointer to the data stored in the `TOmniValue`.

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

1.9 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;
```

1.10 Fluent interfaces

OmniThreadLibrary heavily uses fluent interface[16] approach. Most of the 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`.

[16] http://en.wikipedia.org/wiki/Fluent_interface

```
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;
```

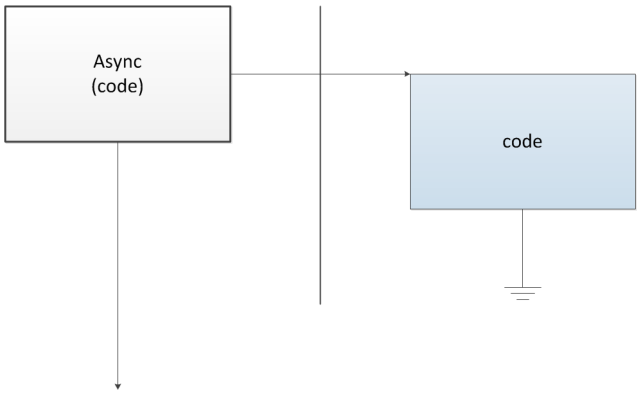
2. High-level multi-threading

Face it – multi-threading programming is hard. It is hard to design a multi-threaded program, it is hard to write and test it and it is *insanely* hard to debug it. To ease this problem, OmniThreadLibrary introduces several pre-packaged multi-threading 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 multi-threaded parts, takes care of synchronisation and so on.

2.1 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. After 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.

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

  Parallel = class
    class procedure Async(task: TProc;
      taskConfig: IOmniTaskConfig = nil); overload;
    class procedure Async(task: TOmniTaskDelegate;
      taskConfig: IOmniTaskConfig = nil); overload;
    ...
  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.

```
Parallel.Async(
  procedure (const task: IOmniTask)
  var
    page: string;
  begin
    HttpGet('otl.17slon.com', 80, 'tutorials.htm', page, '');
    task.Invoke(
      procedure
      begin
        lbLogAsync.Items.Add(Format('Async GET: %d ms; page length = %d',
          [time, Length(page)]))
      end);
  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.

```
const
  WM_RESULT = WM_USER;

procedure LogResult(const task: IOmniTaskControl; const msg: TOmniMessage);
begin
  lbLogAsync.Items.Add(Format('Async GET: %d ms; page length = %d',
    [time, Length(page)]))
end;

Parallel.Async(
  procedure (const task: IOmniTask)
  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*.

2.1.1 Handling exceptions

If the background code raises an 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, an exception will be handled by the application-level filter, which will by default cause a message box to appear.

See also demo `48_OtlParallelExceptions`.

2.2 Async/Await

Async/Await is a simplified version of the *Async* abstraction which mimics the .NET Async/Await[1] mechanism.[2]

Async/Await accepts two parameter less 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;
```

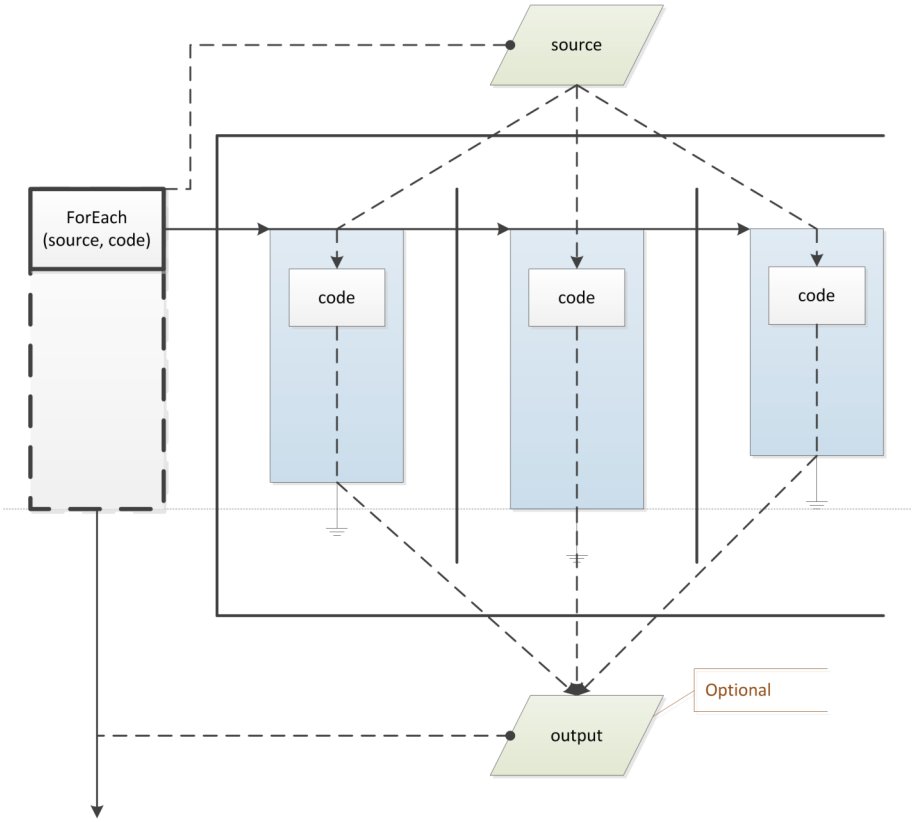
Keep in mind that *Async* is invoked by calling `Parallel.Async` and *Async/Await* by calling `Async`.

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

[1] http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293335.aspx
[2] Read more at http://www.thedelphigeek.com/2012/07/asyncawait-in-delphi.html.

2.3 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 the 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.

2.3.1 Cooperation

The main point of the *ForEach* abstraction is cooperation between the parallel tasks. *ForEach* goes to great lengths to minimize potential 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 the 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 task 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.

2.3.2 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.

2.3.2.1 ... 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.

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

The pseudo-code for *numeric* `ForEach` could be written as

```
i := low;
while ((step > 0) and (i <= high)) or
      ((step < 0) and (i >= high)) do
begin
  // process 'i' in parallel
  if low < high then Inc(i, step)
                else Dec(i, step);
end;
```

2.3.2.2 ... 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])`.

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

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

```
Parallel.ForEach(sl).Execute(
  procedure (const value: TOmniValue)
  begin
    Something(value);
  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.

```
Parallel.ForEach<string>(sl).Execute(
  procedure (const value: string)
  begin
    Something(value);
  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 the VCL enumerators are implemented in this way.

```
class function  ForEach(const enumerable: IEnumerable):
  IOmniParallelLoop; overload;
class function  ForEach(const enum: IEnumerator):
  IOmniParallelLoop; overload;
class function  ForEach(const enumerable: TObject):
  IOmniParallelLoop; overload;
class function  ForEach<T>(const enumerable: IEnumerable):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enum: IEnumerator):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enumerable: TEnumerable<T>):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enum: TEnumerator<T>):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enumerable: TObject):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enum: IEnumerator<T>):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enumerable: IEnumerable<T>):
  IOmniParallelLoop<T>;
```

Support for enumerating over `IEnumerator<T>` and `IEnumerable<T>` was added in version [3.07.7].

2.3.2.3 ... 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 support enumeration, such collection must implement `IOmniValueEnumerable` and `IOmniValueEnumerator` interfaces, which are defined in the *OtlCommon* unit.

```
class function  ForEach(const enumerable: IOmniValueEnumerable):
  IOmniParallelLoop; overload;
class function  ForEach(const enum: IOmniValueEnumerator):
  IOmniParallelLoop; overload;
class function  ForEach<T>(const enumerable: IOmniValueEnumerable):
  IOmniParallelLoop<T>; overload;
class function  ForEach<T>(const enum: IOmniValueEnumerator):
  IOmniParallelLoop<T>; overload;
```

2.3.2.4 ... 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;
```

2.3.2.5 ... 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 a 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 non-trivial 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.

2.3.3 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;
```

2.3.4 IOmniParallelLoop interface

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

```
1   IOmniParallelLoop = interface
2     function  Aggregate(defaultAggregateValue: TOmniValue;
3       aggregator: TOmniAggregatorDelegate): IOmniParallelAggregatorLoop;
4     function  AggregateSum: IOmniParallelAggregatorLoop;
5     procedure Execute(loopBody: TOmniIteratorDelegate); overload;
6     procedure Execute(loopBody: TOmniIteratorTaskDelegate); overload;
7     function  CancelWith(const token: IOmniCancellationToken):
8       IOmniParallelLoop;
9     function  Initialize(taskInitializer: TOmniTaskInitializerDelegate):
10      IOmniParallelInitializedLoop;
11    function  Into(const queue: IOmniBlockingCollection):
12      IOmniParallelIntoLoop; overload;
13    function  NoWait: IOmniParallelLoop;
14    function  NumTasks(taskCount : integer): IOmniParallelLoop;
15    function  OnMessage(eventDispatcher: TObject):
16      IOmniParallelLoop; overload; deprecated 'use TaskConfig';
17    function  OnMessage(msgID: word; eventHandler: TOmniTaskMessageEvent):
18      IOmniParallelLoop; overload; deprecated 'use TaskConfig';
19    function  OnMessage(msgID: word; eventHandler: TOmniOnMessageFunction):
20      IOmniParallelLoop; overload; deprecated 'use TaskConfig';
21    function  OnTaskCreate(taskCreateDelegate: TOmniTaskCreateDelegate):
22      IOmniParallelLoop; overload;
23    function  OnTaskCreate(taskCreateDelegate:
24      TOmniTaskControlCreateDelegate): IOmniParallelLoop; overload;
25    function  OnStop(stopCode: TProc): IOmniParallelLoop;
26    function  OnStop(stopCode: TOmniTaskStopDelegate): IOmniParallelLoop; overload;
27    function  OnStopInvoke(stopCode: TProc): IOmniParallelLoop;
28    function  PreserveOrder: IOmniParallelLoop;
29    function  TaskConfig(const config: IOmniTaskConfig): IOmniParallelLoop;
30  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, the 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 than number of available cores.

If `NumTasks` receives a negative parameter (`< 0`), it specifies the 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.

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.

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

Using this version of `OnStop`, the termination handler can use `task.Invoke` to execute code in the main thread. This, however, requires the *ForEach* abstraction to stay alive until the `Invoke`d 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.

```
var
  loop: IOmniParallelLoop<integer>;

loop := Parallel.ForEach(1, N).NoWait;
loop.OnStop(
  procedure (const task: IOmniTask)
  begin
    task.Invoke(
      procedure
      begin
        // do anything
        loop := nil;
      end);
  end);
loop.Execute(
  procedure (const value: integer)
  begin
    ...
  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.

```
var
  loop: IOmniParallelLoop<integer>;

loop := Parallel.ForEach(1, N).NoWait;
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.

2.3.5 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 re-sort the output before it can be processed further. To ease 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.

2.3.6 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 terrible 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.

```
procedure SumPrimes(var aggregate: TOmniValue; const value: TOmniValue)
begin
  aggregate := aggregate.AsInt64 + value.AsInt64;
end;

procedure CheckPrime(const value: integer; var result: TOmniValue)
begin
  if IsPrime(value) then
    Result := 1;
end;

numPrimes :=
  Parallel.ForEach(1, CMaxPrime)
  .Aggregate(0, SumPrimes)
  .Execute(CheckPrime);
```

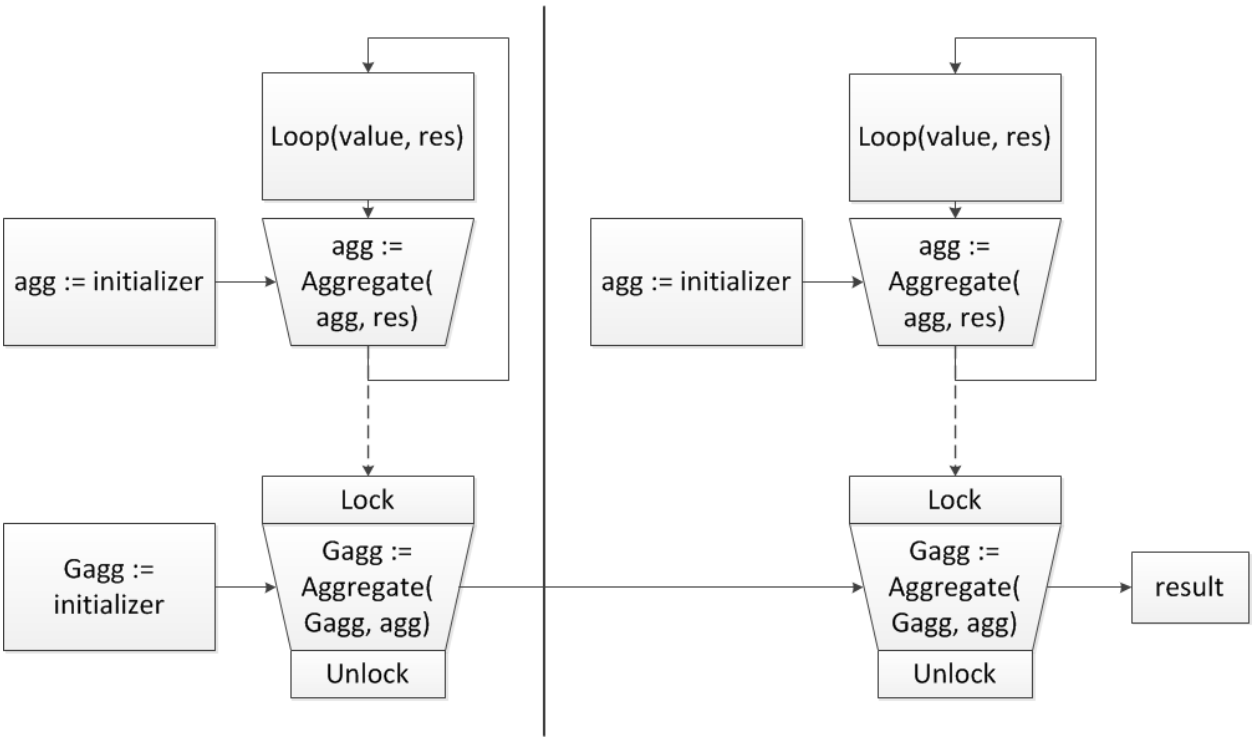
`Aggregate` takes two parameters - the first is the initial value for the aggregate and the second is an aggregation function - a piece of code that will take the current aggregate value and update it with the value returned from the worker task.

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.

```
agg := 0;
result.Clear;
for value := 1 to CMaxPrime do begin
  CheckPrime(value, result);
  if not result.IsEmpty then begin
    SumPrimes(agg, result);
    result.Clear;
  end;
end;
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 the 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;
```

2.3.7 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.

2.3.8 Task initialization and finalization

ForEach implements a mechanism that can be used by worker tasks to initialize 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.

All these 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)
      begin
        if IsPrime(value) then
          taskState.AsInteger := taskState.AsInteger + 1;
      end
    );
end;
```

2.3.9 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.

2.3.10 Examples

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

3. Low-level multi-threading

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.

3.1 Low-level for the impatient

The following code represents the simplest 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.

3.2 Four ways to create a task

Let's examine all four ways of creating a task. The simplest 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 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;
```

3.3 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;
```

```
11      function  GetParam: TOmniValueContainer;
12      function  Invoke(const msgMethod: pointer): IOmniTaskControl; overload;
13      function  Invoke(const msgMethod: pointer;
14        msgData: array of const): IOmniTaskControl; overload;
15      function  Invoke(const msgMethod: pointer;
16        msgData: TOmniValue): IOmniTaskControl; overload;
17      function  Invoke(const msgName: string): IOmniTaskControl; overload;
18      function  Invoke(const msgName: string;
19        msgData: array of const): IOmniTaskControl; overload;
20      function  Invoke(const msgName: string;
21        msgData: TOmniValue): IOmniTaskControl; overload;
22      function  Invoke(remoteFunc: TOmniTaskControlInvokeFunction):
23        IOmniTaskControl; overload;
24      function  Invoke(remoteFunc: TOmniTaskControlInvokeFunctionEx):
25        IOmniTaskControl; overload;
26      function  Join(const group: IOmniTaskGroup): IOmniTaskControl;
27      function  Leave(const group: IOmniTaskGroup): IOmniTaskControl;
28      function  MonitorWith(const monitor: IOmniTaskControlMonitor):
29        IOmniTaskControl;
30      function  MsgWait(wakeMask: DWORD = QS_ALLEVENTS): IOmniTaskControl;
31      function  NUMANode(numaNodeNumber: integer): IOmniTaskControl;
32      function  OnMessage(eventDispatcher: TObject): IOmniTaskControl; overload;
33      function  OnMessage(eventHandler: TOmniTaskMessageEvent): IOmniTaskControl; overload;
34      function  OnMessage(msgID: word; eventHandler: TOmniTaskMessageEvent):
35        IOmniTaskControl; overload;
36      function  OnMessage(msgID: word; eventHandler: TOmniMessageExec):
37        IOmniTaskControl; overload;
38      function  OnMessage(eventHandler: TOmniOnMessageFunction):
39        IOmniTaskControl; overload;
40      function  OnMessage(msgID: word; eventHandler: TOmniOnMessageFunction):
41        IOmniTaskControl; overload;
42      function  OnTerminated(eventHandler: TOmniOnTerminatedFunction):
43        IOmniTaskControl; overload;
44      function  OnTerminated(eventHandler: TOmniOnTerminatedFunctionSimple):
45        IOmniTaskControl; overload;
46      function  OnTerminated(eventHandler: TOmniTaskTerminatedEvent):
47        IOmniTaskControl; overload;
48      function  ProcessorGroup(procGroupNumber: integer): IOmniTaskControl;
49      function  RemoveMonitor: IOmniTaskControl;
50      function  Run: IOmniTaskControl;
51      function  Schedule(const threadPool: IOmniThreadPool = nil {default pool}):
52        IOmniTaskControl;
53      function  SetMonitor(hWindow: THandle): IOmniTaskControl;
54      function  SetParameter(const paramName: string;
55        const paramValue: TOmniValue): IOmniTaskControl; overload;
56      function  SetParameter(const paramValue: TOmniValue):
```

```
57        IOmniTaskControl; overload;
58      function  SetParameters(const parameters: array of TOmniValue):
59        IOmniTaskControl;
60      function  SetPriority(threadPriority: TOTLThreadPriority): IOmniTaskControl;
61      function  SetQueueSize(numMessages: integer): IOmniTaskControl;
62      function  SetTimer(timerID: integer; interval_ms: cardinal;
63        const timerMessage: TOmniMessageID): IOmniTaskControl; overload;
64      procedure SetTimer(timerID: integer; interval_ms: cardinal;
65        const timerMessage: TProc); overload;
66      procedure SetTimer(timerID: integer; interval_ms: cardinal;
67        const timerMessage: TProc<integer>); overload;
68      function  SetUserData(const idxData: TOmniValue;
69        const value: TOmniValue): IOmniTaskControl;
70      procedure Stop;
71      function  Terminate(maxWait_ms: cardinal = INFINITE): boolean;
72      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;
```

3.4 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:

```
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.

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

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

```
FTaskControl.Terminate;
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:

```
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.

```
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;
```

3.5 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 multi-threading 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 the 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 a communication channel usage, see the Communication subsection of simple tasks and `TOmniWorker` tasks sections.

Communication message queue is implemented using the Bounded Queue structure.

3.6 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.

3.7 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 while 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 a 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`.

3.7.1 Bounded Stack

The bounded stack[1] 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] https://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29

```
type
  IOmniStack = interface
    procedure Empty;
    procedure Initialize(numElements, elementSize: integer);
    function  IsEmpty: boolean;
    function  IsFull: boolean;
    function  Pop(var value): boolean;
    function  Push(const value): boolean;
  end;

  TOmniBaseBoundedStack = class(TInterfacedObject, IOmniStack)
  public
    destructor Destroy; override;
    procedure Empty;
    procedure Initialize(numElements, elementSize: integer); virtual;
    function  IsEmpty: boolean; inline;
    function  IsFull: boolean; inline;
    function  Pop(var value): boolean;
    function  Push(const value): boolean;
    property  ElementSize: integer read obsElementSize;
    property  NumElements: integer read obsNumElements;
  end;

  TOmniBoundedStack = class(TOmniBaseBoundedStack)
  public
    constructor Create(numElements, elementSize: integer;
      partlyEmptyLoadFactor: real = CPartlyEmptyLoadFactor;
      almostFullLoadFactor: real = CAlmostFullLoadFactor);
    destructor  Destroy; override;
    function Pop(var value): boolean;
    function Push(const value): boolean;
    property ContainerSubject: TOmniContainerSubject read osContainerSubject;
  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 a 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 the 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.

3.7.2 Bounded queue

The bounded queue[2] 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.

```
1  type
2    IOmniQueue = interface
3      function  Dequeue(var value): boolean;
4      procedure Empty;
5      function  Enqueue(const value): boolean;
6      procedure Initialize(numElements, elementSize: integer);
7      function  IsEmpty: boolean;
8      function  IsFull: boolean;
9    end;
10
11   TOmniBaseBoundedQueue = class(TInterfacedObject, IOmniQueue)
12   public
13     destructor Destroy; override;
14     function  Dequeue(var value): boolean;
15     procedure Empty;
16     function  Enqueue(const value): boolean;
17     procedure Initialize(numElements, elementSize: integer); virtual;
18     function  IsEmpty: boolean;
19     function  IsFull: boolean;
20     property  ElementSize: integer read obqElementSize;
21     property  NumElements: integer read obqNumElements;
22   end;
23
24   TOmniBoundedQueue = class(TOmniBaseBoundedQueue)
25   public
```

[2] https://en.wikipedia.org/wiki/Queue_%28abstract_data_type%29

```
26      constructor Create(numElements, elementSize: integer;
27        partlyEmptyLoadFactor: real = CPartlyEmptyLoadFactor;
28        almostFullLoadFactor: real = CAlmostFullLoadFactor);
29      destructor  Destroy; override;
30      function  Dequeue(var value): boolean;
31      function  Enqueue(const value): boolean;
32      property  ContainerSubject: TOmniContainerSubject read oqContainerSubject;
33    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 the size of the queue element as set in the `Initialize` call.
- `NumElements`

  Returns the 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.

3.7.3 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.

3.7.4 Dynamic queue

The dynamic queue[3] 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.

[3] https://en.wikipedia.org/wiki/Queue_%28data_structure%29

3.7.5 Observing lock-free collections

OmniThreadLibrary data structures support the *observer*[4] 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.

[4] https://en.wikipedia.org/wiki/Observer_pattern

```
  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.

3.7.5.1 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;
```

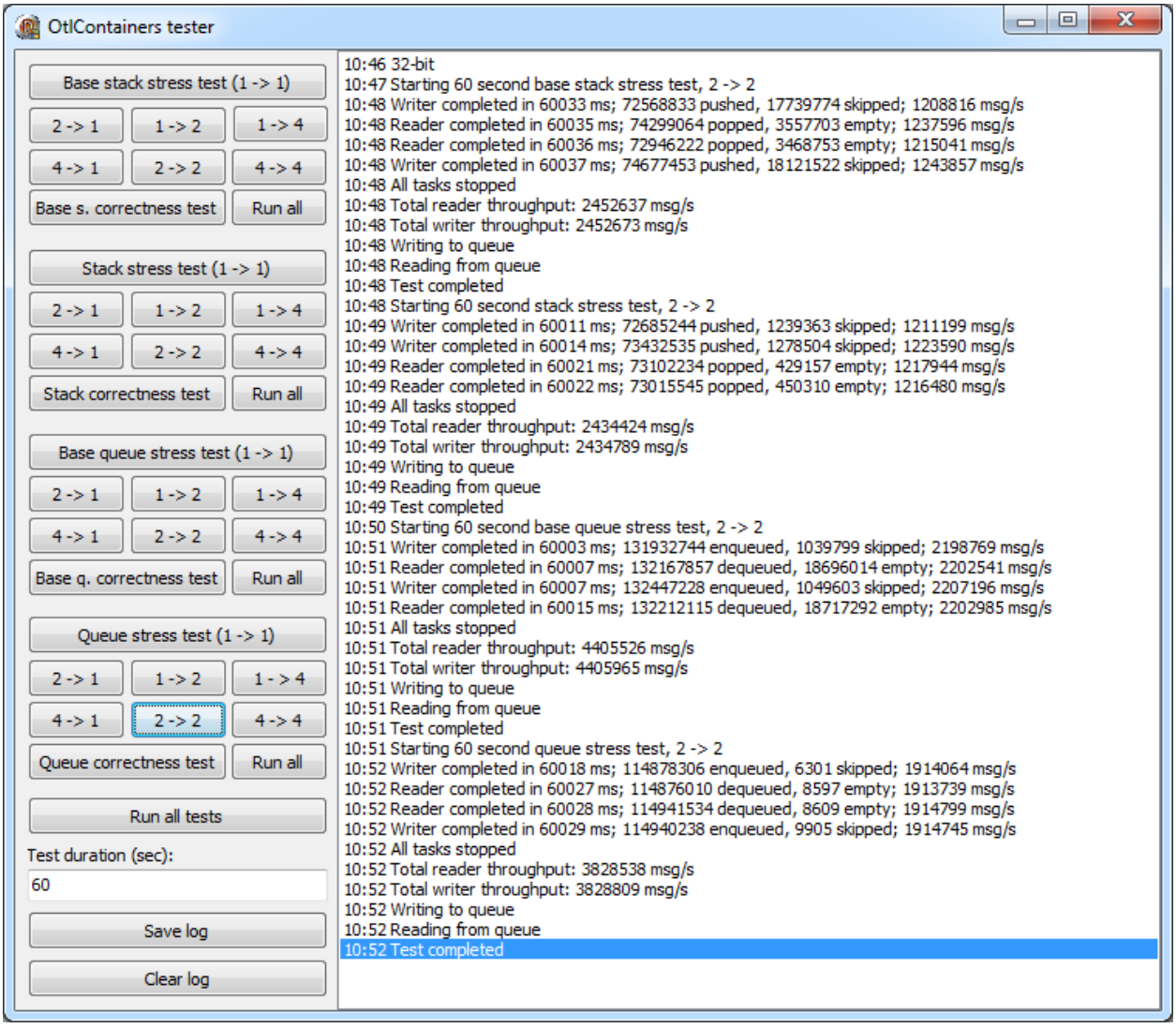
Detach and destroy the observer:

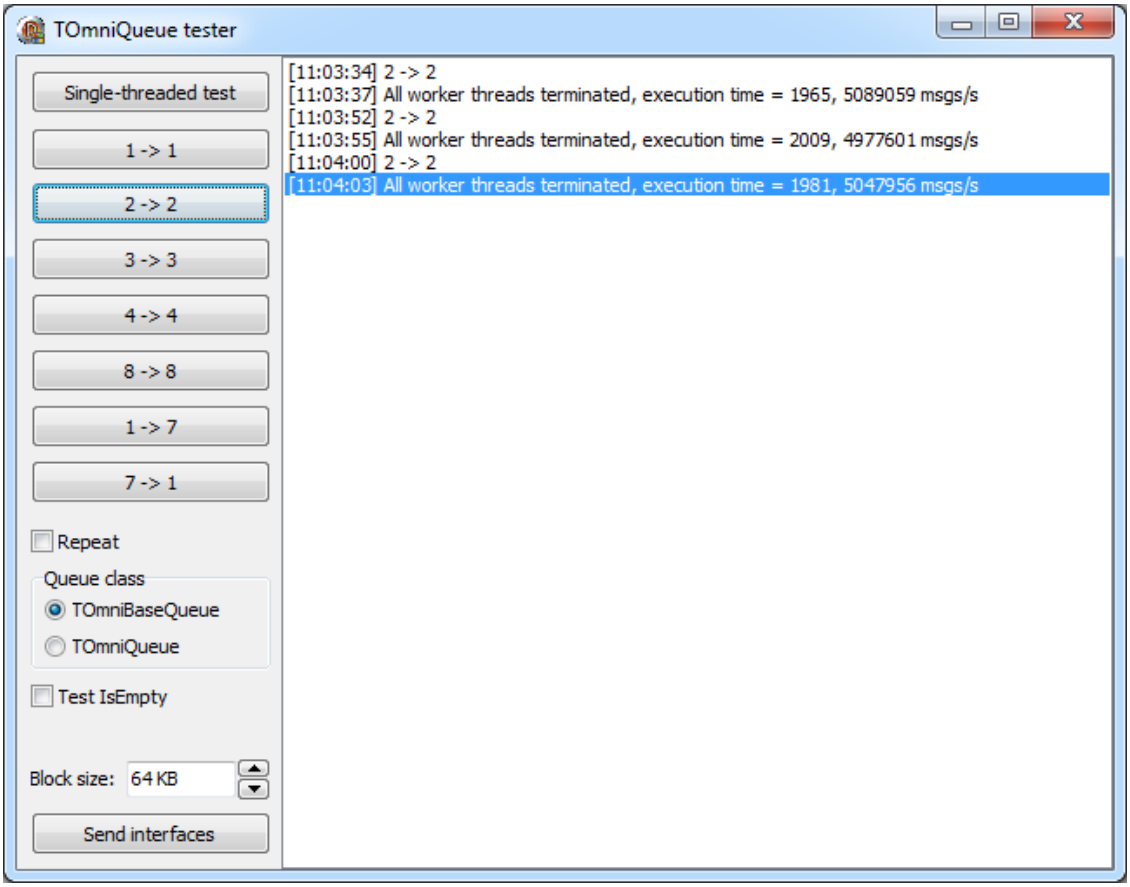
```
FWorker.Output.ContainerSubject.Detach(FObserver, coiNotifyOnAllInserts);
FreeAndNil(FObserver);
DSiDeallocateHWnd(FWindow);
```

3.7.6 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.


OtlContainers tester
Base stack stress test (1 -> 1)
2 -> 1
1 -> 2
1 -> 4
4 -> 1
2 -> 2
4 -> 4
Base s. correctness test
Run all
Stack stress test (1 -> 1)
2 -> 1
1 -> 2
1 -> 4
4 -> 1
2 -> 2
4 -> 4
Stack correctness test
Run all
Base queue stress test (1 -> 1)
2 -> 1
1 -> 2
1 -> 4
4 -> 1
2 -> 2
4 -> 4
Base q. correctness test
Run all
Queue stress test (1 -> 1)
2 -> 1
1 -> 2
1 - > 4
4 -> 1
2 -> 2
4 -> 4
Queue correctness test
Run all
Run all tests
Test duration (sec):
60
Save log
Clear log
10:46 32-bit
10:47 Starting 60 second base stack stress test, 2 -> 2
10:48 Writer completed in 60033 ms; 72568833 pushed, 17739774 skipped; 1208816 msg/s
10:48 Reader completed in 60035 ms; 74299064 popped, 3557703 empty; 1237596 msg/s
10:48 Reader completed in 60036 ms; 72946222 popped, 3468753 empty; 1215041 msg/s
10:48 Writer completed in 60037 ms; 74677453 pushed, 18121522 skipped; 1243857 msg/s
10:48 All tasks stopped
10:48 Total reader throughput: 2452637 msg/s
10:48 Total writer throughput: 2452673 msg/s
10:48 Writing to queue
10:48 Reading from queue
10:48 Test completed
10:48 Starting 60 second stack stress test, 2 -> 2
10:49 Writer completed in 60011 ms; 72685244 pushed, 1239363 skipped; 1211199 msg/s
10:49 Writer completed in 60014 ms; 73432535 pushed, 1278504 skipped; 1223590 msg/s
10:49 Reader completed in 60021 ms; 73102234 popped, 429157 empty; 1217944 msg/s
10:49 Reader completed in 60022 ms; 73015545 popped, 450310 empty; 1216480 msg/s
10:49 All tasks stopped
10:49 Total reader throughput: 2434424 msg/s
10:49 Total writer throughput: 2434789 msg/s
10:49 Writing to queue
10:49 Reading from queue
10:49 Test completed
10:50 Starting 60 second base queue stress test, 2 -> 2
10:51 Writer completed in 60003 ms; 131932744 enqueued, 1039799 skipped; 2198769 msg/s
10:51 Reader completed in 60007 ms; 132167857 dequeued, 18696014 empty; 2202541 msg/s
10:51 Writer completed in 60007 ms; 132447228 enqueued, 1049603 skipped; 2207196 msg/s
10:51 Reader completed in 60015 ms; 132212115 dequeued, 18717292 empty; 2202985 msg/s
10:51 All tasks stopped
10:51 Total reader throughput: 4405526 msg/s
10:51 Total writer throughput: 4405965 msg/s
10:51 Writing to queue
10:51 Reading from queue
10:51 Test completed
10:51 Starting 60 second queue stress test, 2 -> 2
10:52 Writer completed in 60018 ms; 114878306 enqueued, 6301 skipped; 1914064 msg/s
10:52 Reader completed in 60027 ms; 114876010 dequeued, 8597 empty; 1913739 msg/s
10:52 Reader completed in 60028 ms; 114941534 dequeued, 8609 empty; 1914799 msg/s
10:52 Writer completed in 60029 ms; 114940238 enqueued, 9905 skipped; 1914745 msg/s
10:52 All tasks stopped
10:52 Total reader throughput: 3828538 msg/s
10:52 Total writer throughput: 3828809 msg/s
10:52 Writing to queue
10:52 Reading from queue
10:52 Test completed


TOmniQueue tester
Single-threaded test
1 -> 1
2 -> 2
3 -> 3
4 -> 4
8 -> 8
1 -> 7
7 -> 1
Repeat
Queue class
TOmniBaseQueue
TOmniQueue
Test IsEmpty
Block size:
64 KB
Send interfaces
[11:03:34] 2 -> 2
[11:03:37] All worker threads terminated, execution time = 1965, 5089059 msgs/s
[11:03:52] 2 -> 2
[11:03:55] All worker threads terminated, execution time = 2009, 4977601 msgs/s
[11:04:00] 2 -> 2
[11:04:03] All worker threads terminated, execution time = 1981, 5047956 msgs/s

4. 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 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.

4.1 Critical sections

The most useful synchronisation primitive for multi-threaded programming is indubitably the *critical section*[1][2]

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, 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>`.

4.1.1 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] https://en.wikipedia.org/wiki/Critical_section
[2] http://docwiki.embarcadero.com/Libraries/en/System.SyncObjs.TCriticalSection

```
type
  IOmniCriticalSection = interface
    procedure Acquire;
    procedure Release;
    function  GetSyncObj: TSynchroObject;
    property LockCount: integer read GetLockCount;
  end;

function CreateOmniCriticalSection: IOmniCriticalSection;
```

`IOmniCriticalSection` uses `TCriticalSection` internally[3]. 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
```

4.1.2 TOmniCS

Another `TCriticalSection` extension found in the OmniThreadLibrary is the `TOmniCS` record. It allows you to use a critical section by declaring a record in an appropriate place.

Using `TOmniCS`, locking can be as simple as this:

```
uses
  GpLists,
  OtlSync;

procedure ProcessList(const intf: IGpIntegerList);
begin
  //...
end;

var
  lock: TOmniCS;
```

[3] `IOmniCriticalSection` doesn't use `TCriticalSection` directly, but wraps it into a larger object as suggested by Eric Grange.

```
12    intf: IGpIntegerList;
13
14  procedure Test1;
15  begin
16    intf := TGpIntegerList.Create;
17    //...
18    lock.Acquire;
19    try
20      ProcessList(intf);
21    finally lock.Release; end;
22  end;
```

`TOmniCS` is implemented as a record with one private field holding the `IOmniCriticalSection` interface.

```
1   type
2     TOmniCS = record
3     strict private
4       ocsSync: IOmniCriticalSection;
5     private
6       function  GetLockCount: integer; inline;
7       function  GetSyncObj: TSynchroObject; inline;
8     public
9       procedure Initialize;
10      procedure Acquire; inline;
11      procedure Release; inline;
12      property LockCount: integer read GetLockCount;
13      property SyncObj: TSynchroObject read GetSyncObj;
14    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.

```
1   procedure TOmniCS.Acquire;
2   begin
3     Initialize;
4     ocsSync.Acquire;
5   end;
6
7   procedure TOmniCS.Release;
8   begin
9     ocsSync.Release;
10  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;
```

4.1.3 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, a 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;
```

```
16      procedure Release; inline;
17      procedure Free; inline;
18      property Value: T read GetValue;
19    end;
20
21  procedure Locked<T>.Locked(proc: TProc);
22  begin
23    Acquire;
24    try
25      proc;
26    finally Release; end;
27  end;
28
29  procedure Locked<T>.Locked(proc: TProcT);
30  begin
31    Acquire;
32    try
33      proc(Value);
34    finally Release; end;
35  end;
```

4.1.3.1 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` with little work.

```
1   var
2     obj: TGpIntegerList;
3
4   procedure Test4;
5   begin
6     obj := TGpIntegerList.Create;
7     try
8       //...
9       System.TMonitor.Enter(obj);
10      try
11        ProcessObjList(obj);
12      finally System.TMonitor.Exit(obj); end;
13      //...
14    finally FreeAndNil(obj); end;
15  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[4],[5] (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.

4.2 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[6].

On Windows, this is usually accomplished by calling one of the functions from the `WaitForMultipleObjects`[7] family. While they are 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`[8] 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;

    constructor Create; overload;
    constructor Create(const handles: array of THandle); overload;
    destructor  Destroy; override;
    function  MsgWaitAny(timeout_ms, wakeMask, flags: cardinal): TWaitResult;
    procedure SetHandles(const handles: array of THandle);
    function  WaitAll(timeout_ms: cardinal): TWaitResult;
    function  WaitAny(timeout_ms: cardinal; alertable: boolean = false): TWaitResult;
```

[4] http://stackoverflow.com/questions/4856306/tthreadedqueue-not-capable-of-multiple-consumers
[5] http://www.thedelphigeek.com/2011/05/tmonitor-bug.html
[6] https://en.wikipedia.org/wiki/Event_
[7] https://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx
[8] https://msdn.microsoft.com/en-us/library/windows/desktop/ms685061(v=vs.85).aspx

```
22      property Signalled: THandles read FSignalledHandles;
23    end;
```

To use `TWaitFor`, 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 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, use the following approach:

```
var
  wf: TWaitFor;
  info: THandleInfo;

wf := TWaitFor.Create([handle1, handle2]);
try
  if wf.WaitAny(INFINITE) = waAwaited then begin
    for info in wf.Signalled do
      if info.Index = 0 then
        // handle1 is signalled - do something
      else if info.Index = 1 then
        // handle2 is signalled - do something
  end;
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`). It's `wakeMask` and `flags` parameters are the same as the corresponding parameters to the `MsgWaitForMultipleObjectsEx`[9] API.

The use of the `TWaitFor` is shown in demo `59_TWaitFor`.

4.3 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 with no 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.

[9] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684245(v=vs.85).aspx

```
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;
  end;

  TOmniCounter = record
    procedure Initialize;
    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;
  end;

  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).

```
Result := Min(counter, count);
counter := counter - Result;
```

`Take` is used in demo *Parallel Data Production*.

4.4 TOmniAlignedInt32 and TOmniAlignedInt64

Those two records hold *4*-byte (*32* bit) and *8*-byte (*64* bit) values, respectively. These values are suitably aligned so they 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.

```
type
  TOmniAlignedInt32 = record
  public
    procedure Initialize; inline;
    function  Add(value: integer): integer; inline;
    function  Addr: PInteger; inline;
    function  CAS(oldValue, newValue: integer): boolean;
    function  Decrement: integer; overload; inline;
    function  Decrement(value: integer): integer; overload; inline;
    function  Increment: integer; overload; inline;
    function  Increment(value: integer): integer; overload; inline;
    function  Subtract(value: integer): integer; inline;
    class operator Add(const ai: TOmniAlignedInt32; i: integer): cardinal; inline;
    class operator Equal(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
    class operator GreaterThan(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
    class operator GreaterThanOrEqual(const ai: TOmniAlignedInt32; i: integer): boolean;
      inline;
    class operator Implicit(const ai: TOmniAlignedInt32): integer; inline;
    class operator Implicit(const ai: TOmniAlignedInt32): cardinal; inline;
    class operator Implicit(const ai: TOmniAlignedInt32): PInteger; inline;
    class operator LessThan(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
    class operator LessThanOrEqual(const ai: TOmniAlignedInt32; i: integer): boolean;
      inline;
    class operator NotEqual(const ai: TOmniAlignedInt32; i: integer): boolean; inline;
    class operator Subtract(ai: TOmniAlignedInt32; i: integer): cardinal; inline;
    property Value: integer read GetValue write SetValue;
  end;

  TOmniAlignedInt64 = record
  public
    procedure Initialize; inline;
    function  Add(value: int64): int64; inline;
    function  Addr: PInt64; inline;
    function  CAS(oldValue, newValue: int64): boolean;
    function  Decrement: int64; overload; inline;
    function  Decrement(value: int64): int64; overload; inline;
    function  Increment: int64; overload; inline;
    function  Increment(value: int64): int64; overload; inline;
    function  Subtract(value: int64): int64; inline;
    property Value: int64 read GetValue write SetValue;
  end;
```

5. 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 multi-threaded 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 front end for a background task.

- *OmniThreadLibrary and databases*

Using databases from OmniThreadLibrary.

- *OmniThreadLibrary and COM/OLE*

Using COM/OLE from OmniThreadLibrary.

- *Using a message queue with a TThread worker*

Using OmniThreadLibrary's `TOmniMessageQueue` to communicate with a TThread worker.

5.1 Parallel data production

This question comes from StackOverflow[1]. 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.

```
1  if fileSize > CBlockSize then
2    numBytes := CBlockSize
3  else
4    numBytes := fileSize;
5  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.

[1] http://stackoverflow.com/q/7292741/4997

```
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 the 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;
```

```
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 parallel part firstly creates a random generator in each task. 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.

5.2 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.

5.2.1 QuickSort

The first part of this how-to implements a well-known quicksort[2] 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 easy to convert to the multi-threaded form.

```
1  procedure TSequentialSorter.QuickSort(left, right: integer);
2  var
3    pivotIndex: integer;
4  begin
5    if right > left then begin
6      if (right - left) <= CSortThreshold then
7        InsertionSort(left, right)
8      else begin
9        pivotIndex := Partition(left, right, (left + right) div 2);
10       QuickSort(left, pivotIndex - 1);
11       QuickSort(pivotIndex + 1, right);
12     end;
13   end;
14 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 important for the single-threaded version (it only brings a small speedup) but it will help immensely with the multi-threaded version.

Converting this quicksort to a multi-threaded version is simple.

Firstly, we have to create a *Fork/Join* computation pool. In this example, it is stored in a global field.

```
1  FForkJoin := Parallel.ForkJoin;
```

Secondly, we have to adapt the `QuickSort` method.

[2] https://en.wikipedia.org/wiki/Quicksort

```
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 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 `Awaits` 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` 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 the parallel execution is stopped at some level and simple sequential version is used to sort remaining fields.

5.2.2 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.

```
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.

```
function ParallelMax(
  const forkJoin: IOmniForkJoin<integer>;
  left, right: integer): integer;

var
  computeLeft : IOmniCompute<integer>;
  computeRight: IOmniCompute<integer>;
  mid         : integer;

  function Compute(left, right: integer): IOmniCompute<integer>;
  begin
    Result := forkJoin.Compute(
      function: integer
      begin
        Result := ParallelMax(forkJoin, left, right);
      end
    );
  end;

begin
  if (right - left) < CSeqThreshold then
    Result := SequentialMax(left, right)
  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 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.

6. 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 run-time.
- `2_TwoWayHello` : "Hello, World" with bidirectional communication; `TOmniEventMonitor` created in run-time.
- `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 the same as 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 shows 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 communicating with its own worker task.
- 50_OmniValueArray: Wrapping arrays, hashes and records stored 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[1] (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.
- 67_ArrayToCollection: Demonstrates the use of TOmniBlockingCollection.FromArray and FromRange.

[1] http://docwiki.embarcadero.com/Libraries/en/System.Threading.TParallel.For

7. C. 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 multi-threaded 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 a message queue with a TThread worker

  Using `TOmniMessageQueue` to communicate with a `TThread`-based worker.
- `twofish`

  OmniThreadLibrary and databases

  Using databases from OmniThreadLibrary.

8. D. 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.

8.1 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.

8.2 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);
```

8.3 Pool notifications

Pool notifications allow your code to be notified when a thread pool is being created or destroyed. This allows the application to modify 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);
```