DDDebug

Version 1.0.7

Manual

www.ddobjects.de

stefan@ddobjects.de

Table of Content

Introduction

DDDebug is a collection of debugging tools which contains several modules: a memory profiler, a thread viewer, a module viewer and an enhanced exception handler. DDDebug can be integrated easily into your Delphi projects: either using its integrated graphical user interface, its API or - based on the DDObjects framework - as integrated lightweight server application which enables remote access using a thin client application.

The memory profiler traces memory allocations, recognizes the type of allocation - whether it has been done because of creating an object, string, array or record - and provides up to date information about memory usage and statistics. Unlike other solutions it does support packages, is independent of any specific compiler switches and provides immediate feedback.

In a nutshell, the following features are implemented:

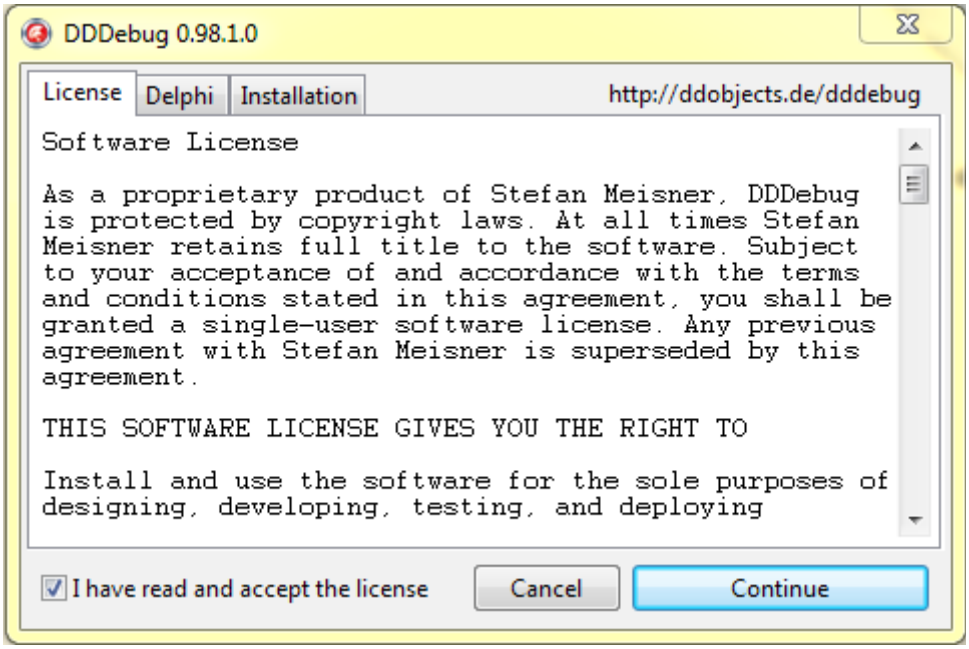
- recognizes objects, strings, arrays and records
- supports packages
- supports profiling of multiple threads
- has call stack retrieval
- can use standard MAP files
- is independent of compiler switches
- contains integrated GUI for monitoring and debugging
- is easy to use; almost no special requirements
- has support for wait chain traversal
- offers detection of thread deadlocks
- offers detection of premature freed memory
- and much more...

System Requirements

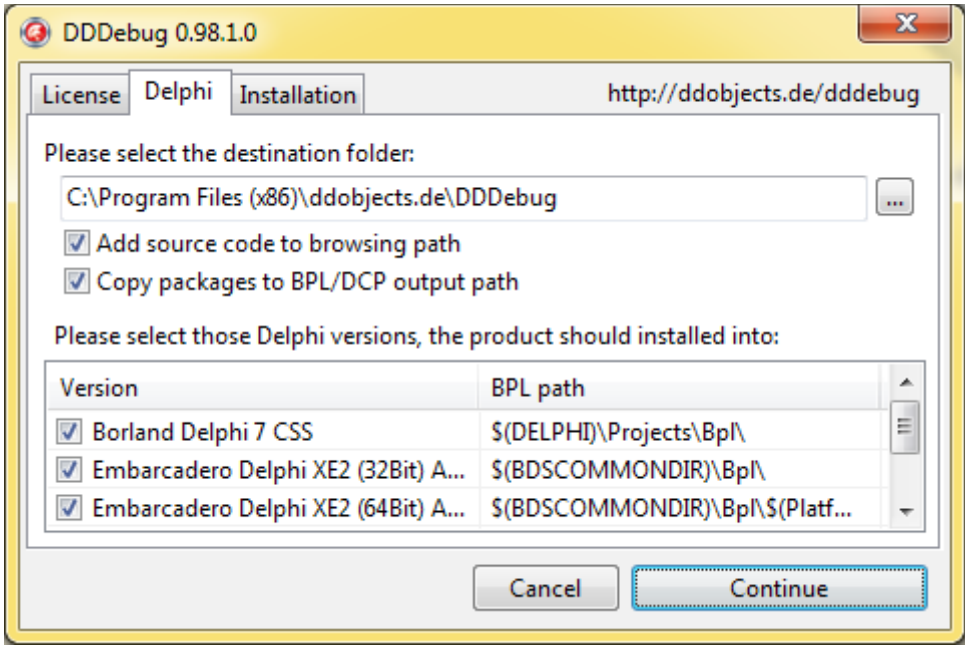
DDDebug can be used with Delphi 5 to Delphi Rio running on Windows XP and above. Some of its features, for example the wait chain traversal, are only available on Windows Vista and above. Where applicable, these limitations are mentioned within the next sections.

Setup of DDDebug

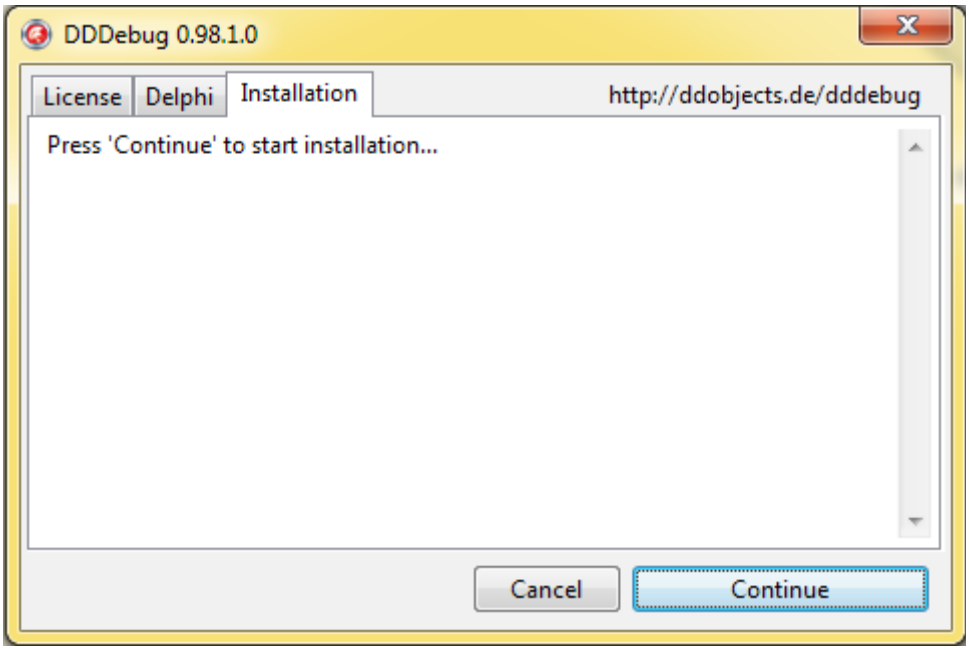
Execute DDDebugSetup.exe and follow the instructions as shown in the next couple of screenshots. Please read and accept the licence and click on the button “Continue” which will lead you to the next screen.



Select the destination, DDDebug should be installed into, select if you would like to add the sources to Delphis browsing path and copy packages to Delphis default output path (both options are selected by default). Finally select the Delphi versions DDDebug should be installed into. Click on the button “Continue” which will lead you to the next screen.



Start the installation by clicking on “Continue”. The view will display messages during installation progress. In case of an error you can continue with the manual installation described in the next chapter.



The setup is based on my free Delphi Component Installer which is available at http://ddobjects.de/free-software

> *“During development of DDDebug, the need for a small setup emerged which compiles and adds packages and search pathes to the Delphi IDE. Therefore I have created a small and lightweight Delphi Component Installer supporting Delphi 5 to Delphi Rio. The code can be downloaded and used freely, integrated into own setup projects or used for other purpose. The zip file contains the source code as well as a working sample.*
> *To compile and use the source you need at least Delphi XE. Users of Delphi XE may need to add a custom ZIP implementation or use the prepared integration of TurboPower Abbrevia.”*

Manual Installation

Execute the setup as described within the prior chapter without selection of any Delphi version. The DCU and other files are contained within the folder "Lib" which contains several sub folders; one for each supported Delphi version. Add the appropriate sub folder to your Delphi library path.

Notes for Delphi 7

If you have installed service pack 1 for Delphi 7, you need to use the DCU files contained within the folder "Lib\Delphi71" otherwise add the folder named "Lib\Delphi7" to your library path.

Notes for Delphi XE2 and above

As Delphi XE2 and above supports 32 as well as 64bit, the appropriate precompiled DCU files can be found in different sub folders: "Lib\DelphiXE2\32Bit" and "Lib\DelphiXE2\64Bit" and accordingly.

Notes for the full Version (incl. Sources)

If you want to rebuild the source and packages using Delphi 7 with service pack 1 installed, please open the file "DDDebug.inc" contained within the folder "Source", navigate to the section shown below and ensure that $DEFINE Delphi71 is activated (remove the . in front of $DEFINE)

```
{$IFDEF VER150}
 {$DEFINE Delphi7}
 {.$DEFINE Delphi71} <-- remove the . to activate
 {$DEFINE Delphi5up}
 {$DEFINE Delphi6up}
 {$DEFINE Delphi7up}
{$ENDIF}
```

The package files can be found in the sub folder "packages" contained within the folder "Source", one for each Delphi version as listed here:

- DDDebugD5.dpk
- DDDebugD6.dpk
- DDDebugD7.dpk
- DDDebugD2005.bdsproj
- DDDebugD2006.bdsproj
- DDDebugD2007.dproj
- DDDebugD2009.dproj
- DDDebugD2010.dproj
- DDDebugDXE.dproj
- DDDebugDXE2.dproj (32 and 64bit)
- DDDebugDXE3.dproj (32 and 64bit)
- DDDebugDXE4.dproj (32 and 64bit)
- DDDebugDXE5.dproj (32 and 64bit)
- DDDebugDXE6.dproj (32 and 64bit)
- DDDebugDXE7.dproj (32 and 64bit)
- DDDebugDXE8.dproj (32 and 64bit)
- DDDebugXSeattle.dproj (32 and 64bit)
- DDDebugXBerlin.dproj (32 and 64bit)
- DDDebugXTokyo.dproj (32 and 64bit)
- DDDebugXRio.dproj (32 and 64bit)

Before rebuilding the sources and packages you might need to execute the batch file "CopyDFM.bat" which prepares the folder "Lib", copying all needed RES-, DFM- and DCR-files.

Integration into IDE

The design time package files can be found in the sub folder "packages" contained within the folder "Source", one for each Delphi version as listed here:

- dclDDDebugD6.dpk
- dclDDDebugD7.dpk
- dclDDDebugD2005.bdsproj
- dclDDDebugD2006.bdsproj
- dclDDDebugD2007.dproj
- dclDDDebugD2009.dproj
- dclDDDebugD2010.dproj
- dclDDDebugDXE.dproj
- dclDDDebugDXE2.dproj
- dclDDDebugDXE3.dproj
- dclDDDebugDXE4.dproj
- dclDDDebugDXE5.dproj
- dclDDDebugDXE6.dproj
- dclDDDebugDXE7.dproj
- dclDDDebugDXE8.dproj
- dclDDDebugXSeattle.dproj
- dclDDDebugXBerlin.dproj
- dclDDDebugXTokyo.dproj
- dclDDDebugXRio.dproj

Open and build the appropriate package and install it afterwards. Please note, that there's no build configuration for 64bit ,as design time packages, which are loaded into the IDE, are always 32bit.

In Delphi 2009 and up, DDDebug is integrated within the IDE. This offers memory tracking and profiling during design time e.g. when developing delphi components.

Please note that there's no IDE integration for Delphi 5 available

Alternative Memory Managers

Until now I used and tested DDDebug with Delphi's standard memory manager mainly. Further I tested DDDebug using the Fast Memory Manager (known as FastMM) which is Delphi's standard memory manager since Delphi 2007. Executing the unit tests showed, that DDDebug works equally well with ScaleMM and TopMM and, due to its architecture, should also work with any other memory manager.

If you are using an alternative memory manager and experience major problems (likely Access Violations) please let me know.

Please note that ReportMemoryLeaksOnShutDown will conflict with DDDebug as some of its structures are freed lately; however, the integrated memory profiler completely replaces this functionality, storing these memory leaks in a configurable log file.

Integrating with EurekaLog and MadExcept

Some users of DDDebug use EurekaLog or MadExcept for advanced exception handling and DDDebug for memory profiling. EurekaLog and MadExcept do implement alternative ways to retrieve the names of methods and includes necessary data into the executable files while DDDebug uses standard map files. All methods do have their pros and cons. However, if using EurekaLog or MadExcept, DDDebug can use those when resolving the names of methods so no map files need to be available or distributed.

In order to use EurekaLog or MadExcept to retrieve names of methods, all you need to do is to include a single unit within your application and invoke a simple initialization. Both units are stored within the sub folder “Custom”.

To use EurekaLog, include the unit **DDDEurekaLog** and call

```
TDDDEurekaLogHandler.Initialize;
```

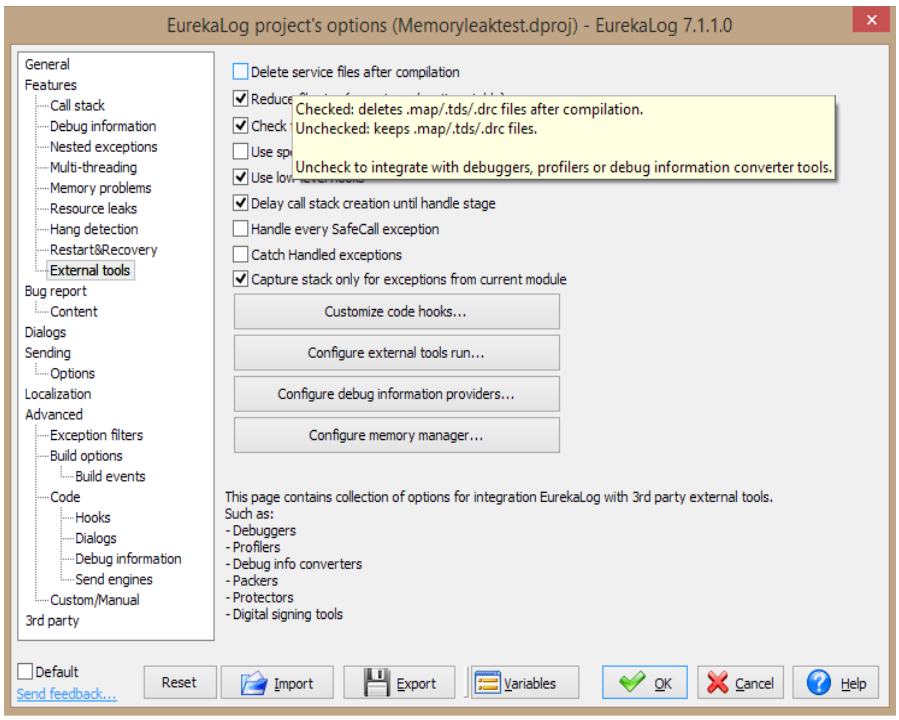
To use MadExcept, include the unit **DDDMadExcept** and call

```
TDDDMadExceptHandler.Initialize;
```

Other 3rd party libraries most likely can be integrated in a similar way. Both units can be used as a template and adjusted accordingly.

DDDebug and EurekaLog

In case, you are using DDDebug in combination with EurekaLog and do not want to use the approach described above, please ensure that EurekaLog does not delete the Delphi map file after processing. To do so, ensure that “*Delete service files after compilation*” is not checked as shown in the screenshot to the right.



Changes and Updates

1.0.7	Support for Delphi Rio added
1.0.6	Support for Delphi Berlin and Tokyo added Minor fixes (Filter was not activated by UI) Sample collecting statistics (useful for services etc.) AV after closing snapshot fixed
1.0.5	Extended filter UI Support for Delphi Seattle added
1.0.4	Support for Delphi XE8 added
1.0.3	Fixed a rare Access Violation within the user interface which occurred when profiling all types of memory with multiple threads involved.
1.0.2	ZeroOnRelease (optional): fills freed memory with 0 for easier debugging Bug fixed: main form always in front in some older Delphi versions Optional integration of EurekaLog and MadExcept (and other 3rd party libraries) Fixed issue with resource file in Demo which could not be build :-|
1.0.1	Increase of performance (up to 100%) if no runtime packages involved Global hotkey Strg+Alt+D opens DDDebug Configurable depth of call stack (default value of 10 for better performance) Incremental search in list views (Strg+F) Fixed crash in Demo with Delphi 5
1.0.0	Fixed total crash in Delphi 5 Fixed UI issues UI tests using Sikuli added Functions Remove and RemoveAll Statistics enhanced Minor cleanup and enhancements Support for Delphi XE7

0.99.1 Added support for Delphi XE6
Dialog sizes and positions persistent
Setup (XE2 and up) fixed

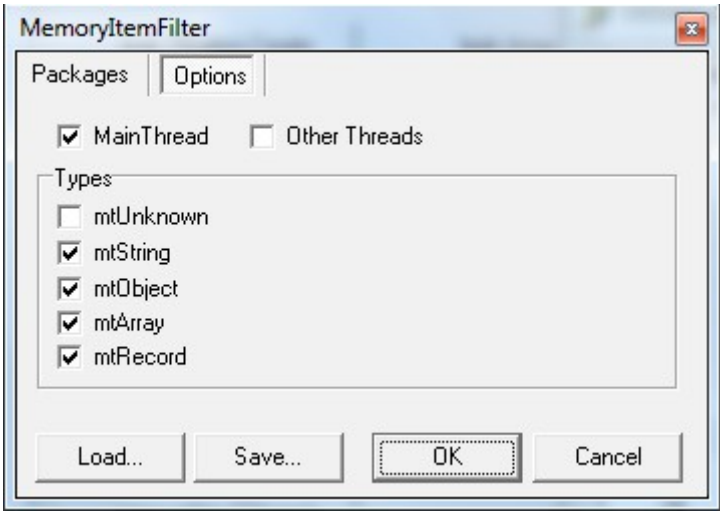
0.99 Exceptionhandler enhanced and elaborated
Thread exception handling enhanced and elaborated
Thread viewer enhanced (does not depend on external DLL any more)
New feature: integration of profiler into Delphi IDE
New feature: module handler to list devices and device drivers
New feature: filtering of anonymous methods (see “Anonymous Methods”)
New feature: tracing of “short living” objects (see “Memory Statistics”)
Added support for Delphi XE4 and XE5
Added client application for remote and service profiling (see “Remote Profiling”)
Fixed a bug for properties of type kind tkClassRef, tkPointer, tkProcedure
64bit: fixed a rare Access Violation in DDDCore
64bit (XE3 and above): fixed a declaration in DDDImageHlp

0.98.1 Revised and updated manual (this document)
Added filter to trace allocations of specific threads only
Integration into IDE (open unit in IDE)
64bit: fixed an issue where a pointer could get truncated
64bit: fixed an issue where method name was corrupted
64bit: fix in GetDataString (for Strings), unit test added
Fixed an Access Violation in Thread Viewer and Wait Chain Traversal
Alternative self contained Setup for Delphi 5 to XE3
DDDebug main form usable while modal dialogs being shown

Introduction

A small demo has been provided within the folder "Demo\Local". If you start the application you'll notice a window labeled "DDDebug". Check the check boxes labeled "Active" and "Live". The latter will display the allocations as they took place, so to speak in real time. Click on the buttons labeled "leak TButton" and “leak Form”. You should notice that the list view will show new entries on each click. Double-click on an entry to display its details.

In order to trace strings, records, arrays and threads you need to configure the profiler accordingly. The form provides a pop up menu in order to do so. Select the menu item labeled "Filter" to display and configure these options. On the tab sheet labeled "Options" check all check boxes, click on "Ok" and confirm the next dialog.



Click on all the other buttons, e.g. "leak String", and notice the new entries within the list view which appear on each allocation. Double-click on an entry to display its details, play around. The other tab sheets, "Statistics", "ThreadViewer", "ExceptionHandler" and "ModuleViewer" are described within the next sections.

Please note that the folder contains different project files as listed here:

DDDDemo.dpr	Delphi 5, 6 and 7
DDDDemo.bdsproj	Delphi 2005 and 2006
DDDDemo.dproj	Delphi 2007
DDDDemo2009.dproj	Delphi 2009, 2010 and XE
DDDDemoXE2.dproj	Delphi XE2 and above (32bit and 64bit)

Note for users of Delphi 5: due to an issue including a manifest file and using list views, which leads to strange Access Violations, please open the project source file and comment out or remove the following line:

```
{$R 'DDDDemo.res' 'DDDDemo.rc'}
```

DUnit Tests

A couple of unit tests are provided within the folder "DUnit". The DUnit tests cover various aspects of DDDebug, e.g. using its API to access the memory profiler and module handler, using the exception handler, the detection of premature freed memory and thread deadlocks. So to speak, the unit tests can be seen as part of the documentation ;-)

Please note that the folder contains different project files as listed here:

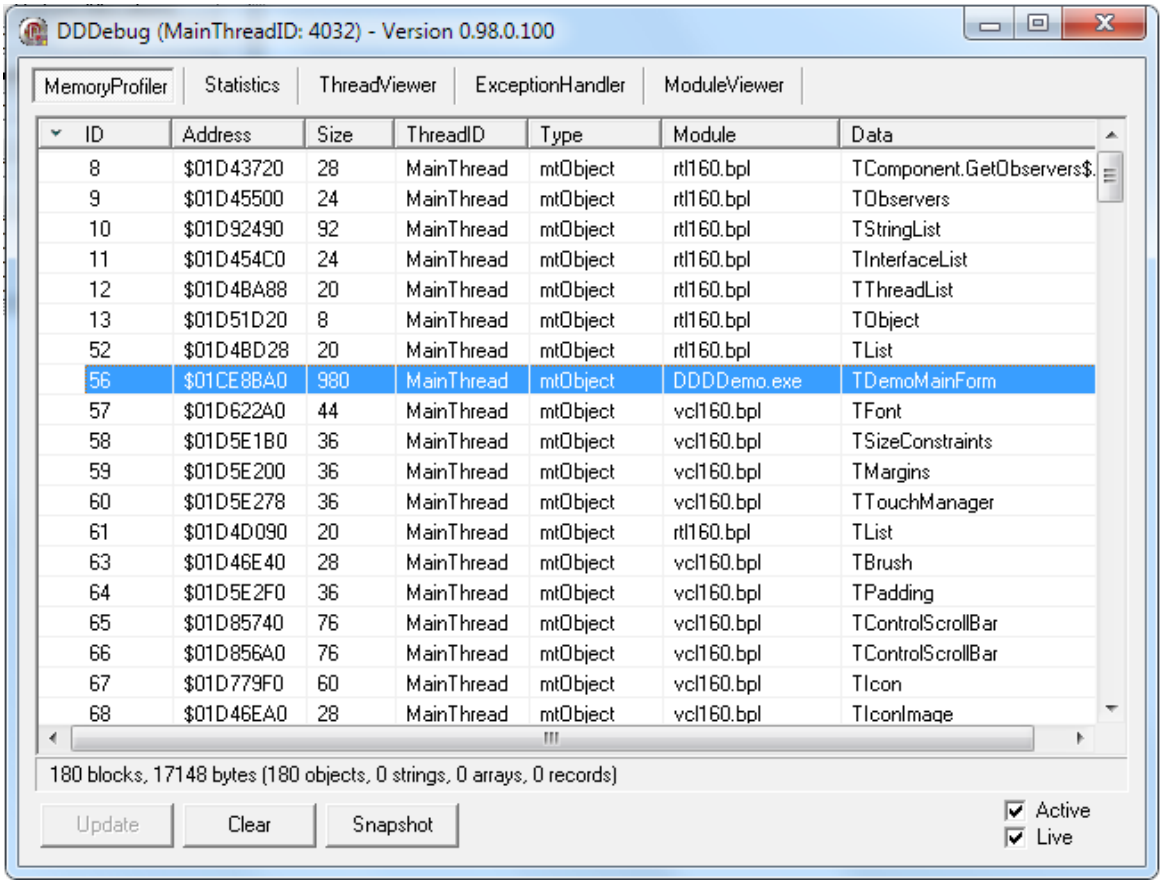
DDDebugTests.dpr	Delphi 5, 6 and 7
DDDebugTests.bdsproj	Delphi 2005 and 2006
DDDebugTests.dproj	Delphi 2007
DDDebugTests2009.dproj	Delphi 2009, 2010 and XE
DDDebugTestsXE2.dproj	Delphi XE2 and above (32bit and 64bit)

The Memory Profiler

Using DDDebug in your projects is easy: adding the unit DDDMainForm to the uses clause of your project is almost all you have to do in order to integrate it into your project. It is recommended to use the applied GUI to monitor memory allocations. The GUI gives you full control about almost any aspect of the memory profiler. Create the form somewhere in your code as shown next:

```
TDDDMainForm.GetInstance.Show;
```

To take full advantage of the memory profiler your project should be built with MAP files. In order to request Delphi generating those files during the next build select “Project" then "Options”, switch to the tabsheet labeled “Linker” and select “Detailed” within the box labeled “Map File”.



The form consists of a list view displaying allocated blocks of memory, along with some details for each entry, a label summarizing the list views entries as well as some check boxes and buttons comprising the most frequent used functions. The items of the list view can be ordered in ascending and descending order by clicking on the appropriate column header. The current order is indicated by a small arrow as shown for the column “ID” in the above screenshot.
Select the check box “Activate” to activate resp. deactivate the memory profiler as well as the check box “Live” to update the list view automatically. For multi threaded and/or time consuming

operations, it is recommended to deactivate this feature. If you do not select “Live”, you have to update it manually by clicking the button “Update”. The button “Clear” discards all entries.

You can see the address of each item, its size, the ID of the thread having allocated the specific block of memory, the kind of memory allocation, the name of the package containing the specific type as well as some additional information specific to each type.

As you can see, the profiler distinguishes between several kinds of memory allocations:

- mtObject: an object has been allocated
- mtString: a string has been allocated
- mtArray: an array has been allocated
- mtRecord: a record has been allocated
- mtUnknown: type could not be recognized

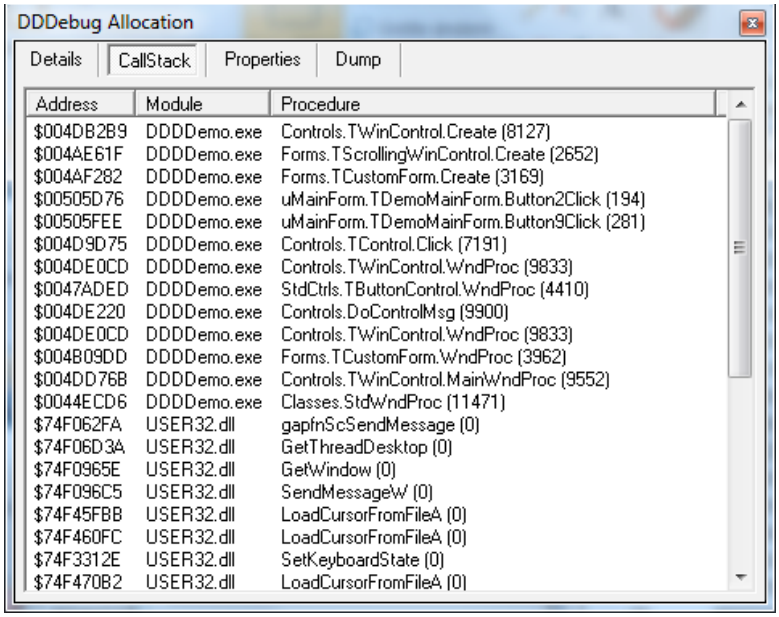
The content of the column “Data” depends on the recognized type:

- the class name of the object being allocated
- the reference count, length and content of the string
- the reference count and length of the array
- the name of the record type being allocated

The column “Module” displays the name of the package, which contains the specific type. For example, you can see on the screen shot above that TFont is defined within the package vcl160.bpl while TList is defined within the package rtl160.bpl. TDemoMainForm is not defined within a separate package but within the main-executable.
Please note that arrays and strings (mtArray and mtString) as well as unrecognized allocations (mtUnknown) are always reported as being defined within the main executable.

Double click on any item for detailed information of the selected entry. A new dialog “DDDebug Allocation” will be displayed which provides several tab sheets containing in-depth-information. While the first one “Details” displays the same information already shown within the list view the other tab sheets contain much more details regarding the selected block of memory. Please notice that the visibility of these tab sheets depends on specific conditions, which will be explained as suitable.

The tab sheet “CallStack” is probably of most interest and is shown within the next screen shot. One can see which flow of code has been executed leading to the allocation of the selected item. The address is always available, while the identification of the names of the procedures requires the above mentioned MAP files to be present. Call stacks are by default generated for objects only but you can select any kind by selecting the appropriate item within the list views context menu.



Double click on an entry within the call stack to open the Delphi unit which contains the selected procedure.

The tab sheet “Properties” is only available for objects where Delphi has generated run time type information (RTTI). This includes classes descending from TPersistent as well as all classes directly or indirectly inheriting from a class which has been compiled with RTTI (search for {$M+} in your Delphi help for further information). The tab sheet contains a list of all published properties with their current value.

Please note that this version of DDDebug still uses the “old” RTTI. The next major version will use the enhanced RTTI features as introduced in Delphi 2009 to supply even more information, allowing to change the property values of the inspected object at run time.

If the selected property itself contains a reference to an object and the referenced object has been traced by the memory profiler you can display its details by double-clicking on the property.
The last tab sheet “Dump” provides a view of the allocated block of memory, similar to a Hex-viewer. The dialog can be closed by clicking on the button in the upper right corner or by pressing the ESC key. However: if you leave it open, the dialog will update itself as soon as you double-click on another item within the list view.

Profiling multi threaded Applications

By default the memory profiler only traces allocations of the main thread.

To trace all threads switch to DDDebug's main form, right click to show up it’s context menu and select the menu item labeled "Filter". Within the next window select the check box "All Threads" and confirm the selection as well as the following notice.

Working with Filters

Most likely, the list of traced items will contain many thousand entries. To achieve a better overview and focus on some concrete classes and parts of the application, you can configure the above mentioned filter which will trace items of a certain kind only. The first tab sheet “Packages” contains a list of all packages, which make up your project, as well as the main executable. Select those packages whose types should be traced.

It should be reinforced that arrays, strings and unrecognized types are assigned to the main executable. Therefore you should not deselect it if you want to trace items of that kind.

The next tab sheet “Options” allows selecting those kinds of memory allocations you would like to be included within the trace as well as offering the opportunity to restrict the trace to allocations that have been caused by the main thread of the process.

Apply your configured filter by pressing the “OK” button. In case the configuration is not reasonable, you will be pointed out to the erroneous settings. You can cancel your changes by pressing the button “Cancel”.

You can save your settings and reload them for later use by pressing the appropriate buttons “Load” and “Save” which will display an ordinary “File Open” resp. “File Save” dialog.

Please note that the performance of the memory profiler suffers from the number of items being traced. If you are tracing objects only, the loss of performance will not be very significant. However: tracing items of mtUnknown and mtString will severely harm the performance as even the smallest allocation - even if it does exist for a very short time only, as it’s e.g. very often the case for strings - needs to be accounted for.

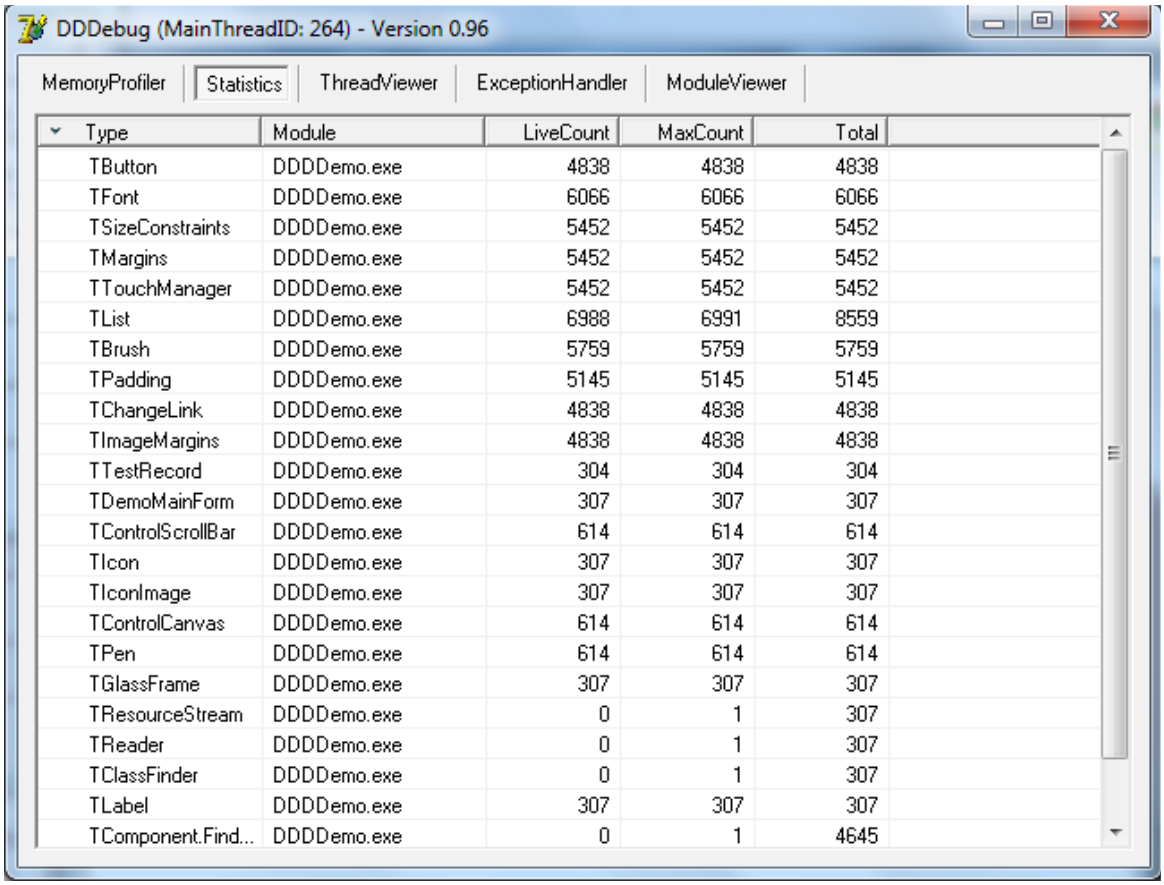
More included filters are described within the section “More about Filters”.

Please note the limitations of the trial version.

The Memory Statistics

The view within the tab "Statistics" displays a list of all records and classes which have been allocated, showing how many instances of each item have been created, how many instances of each type are currently allocated as well as the peak number of allocations of each type.

This view is useful to identify growing allocations quickly and can be helpful in optimizing the systems behavior as e.g. Action updates might be implemented in a suboptimal manner, allocating and deallocating memory on each event (see “Tracing of Short Living Objects”).



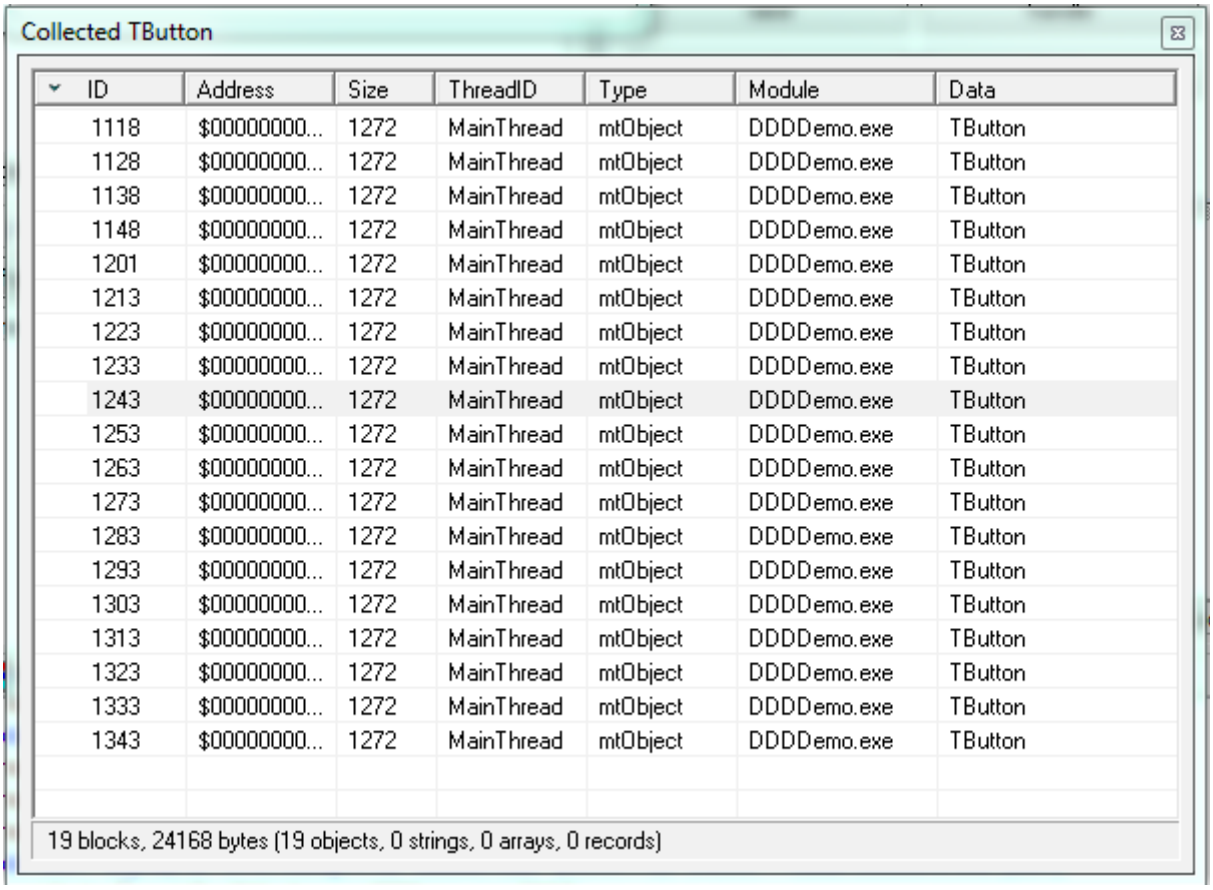
Type	Module	LiveCount	MaxCount	Total
TButton	DDDDemo.exe	4838	4838	4838
TFont	DDDDemo.exe	6066	6066	6066
TSizeConstraints	DDDDemo.exe	5452	5452	5452
TMargins	DDDDemo.exe	5452	5452	5452
TTouchManager	DDDDemo.exe	5452	5452	5452
TList	DDDDemo.exe	6988	6991	8559
TBrush	DDDDemo.exe	5759	5759	5759
TPadding	DDDDemo.exe	5145	5145	5145
TChangeLink	DDDDemo.exe	4838	4838	4838
TImageMargins	DDDDemo.exe	4838	4838	4838
TTestRecord	DDDDemo.exe	304	304	304
TDemoMainForm	DDDDemo.exe	307	307	307
TControlScrollBar	DDDDemo.exe	614	614	614
TIcon	DDDDemo.exe	307	307	307
TIconImage	DDDDemo.exe	307	307	307
TControlCanvas	DDDDemo.exe	614	614	614
TPen	DDDDemo.exe	614	614	614
TGlassFrame	DDDDemo.exe	307	307	307
TResourceStream	DDDDemo.exe	0	1	307
TReader	DDDDemo.exe	0	1	307
TClassFinder	DDDDemo.exe	0	1	307
TLabel	DDDDemo.exe	307	307	307
TComponent.Find...	DDDDemo.exe	0	1	4645

Double click on an entry to see a list view displaying all current allocated items of the selected type, their values, properties and call stacks as described at the beginning of this section.

Tracing of Short Living Objects

As mentioned above, the tracing of short living objects, which are created and destroyed periodically as it's often the case within action update events, can be observed within the Statistics view. However, those allocations can not be inspected within the main view of the profiler as it only displays the current state of allocations.

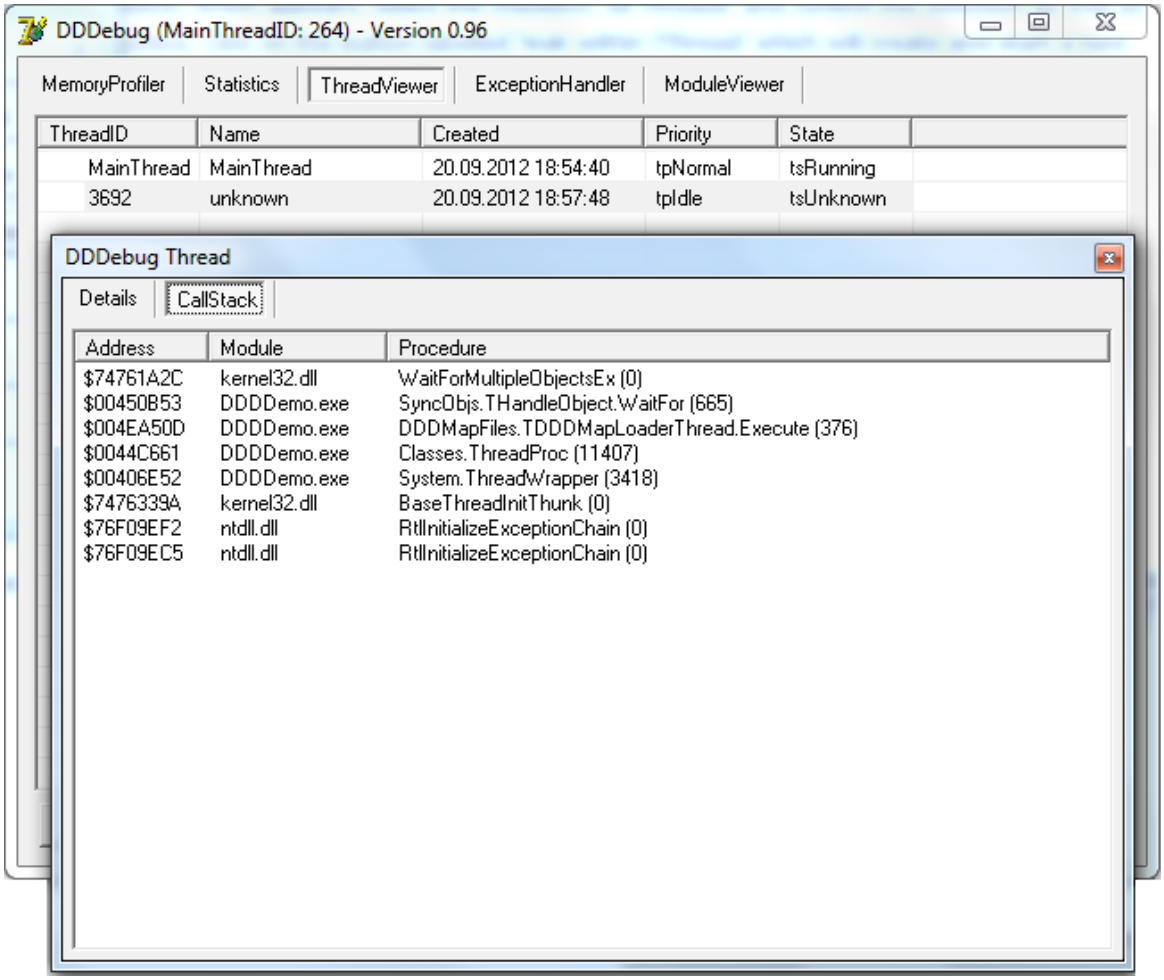
DDDebug provides the opportunity to collect a list of such short living objects via the Memory Statistics. Select the type which allocations you want to trace within the Statistics View and select “Collect” within the popup menu. The next screen shot displays such a collection for the type TButton. All allocated objects of this type will be collected and displayed within the forms listview, regardless whether the object is still alive or has been destroyed in the meantime.



Each object can be inspected and displayed by double clicking on one of the rows of the list view.

The Thread Viewer

The view within the tab "ThreadViewer" displays a list of all threads within the process. Double-click on an entry to retrieve details of the selected thread as well as the call stack which reveals what the thread is doing right now - as shown within the next screen shot.

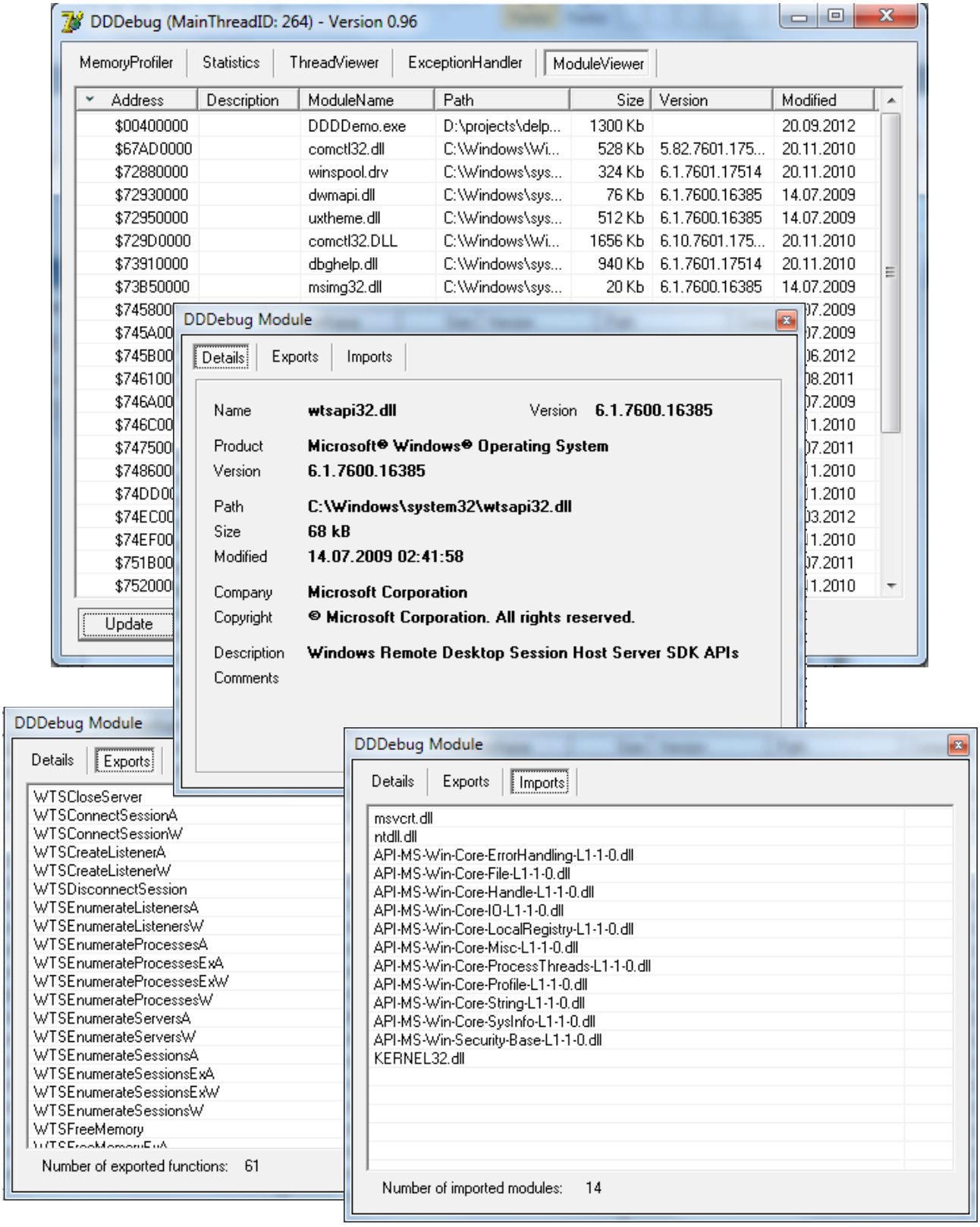


The selected thread is part of DDDebug and responsible for reading and parsing the MAP files which are needed to retrieve the method names for call stacks of memory allocations, exceptions and threads. As you can see due to these names, the thread is waiting for an event (THandleObject.WaitFor which invokes WaitForMultipleObjectsEx in kernel32.dll).

More features of DDDebug concerning threads are described within the section "Finding deadlocks".

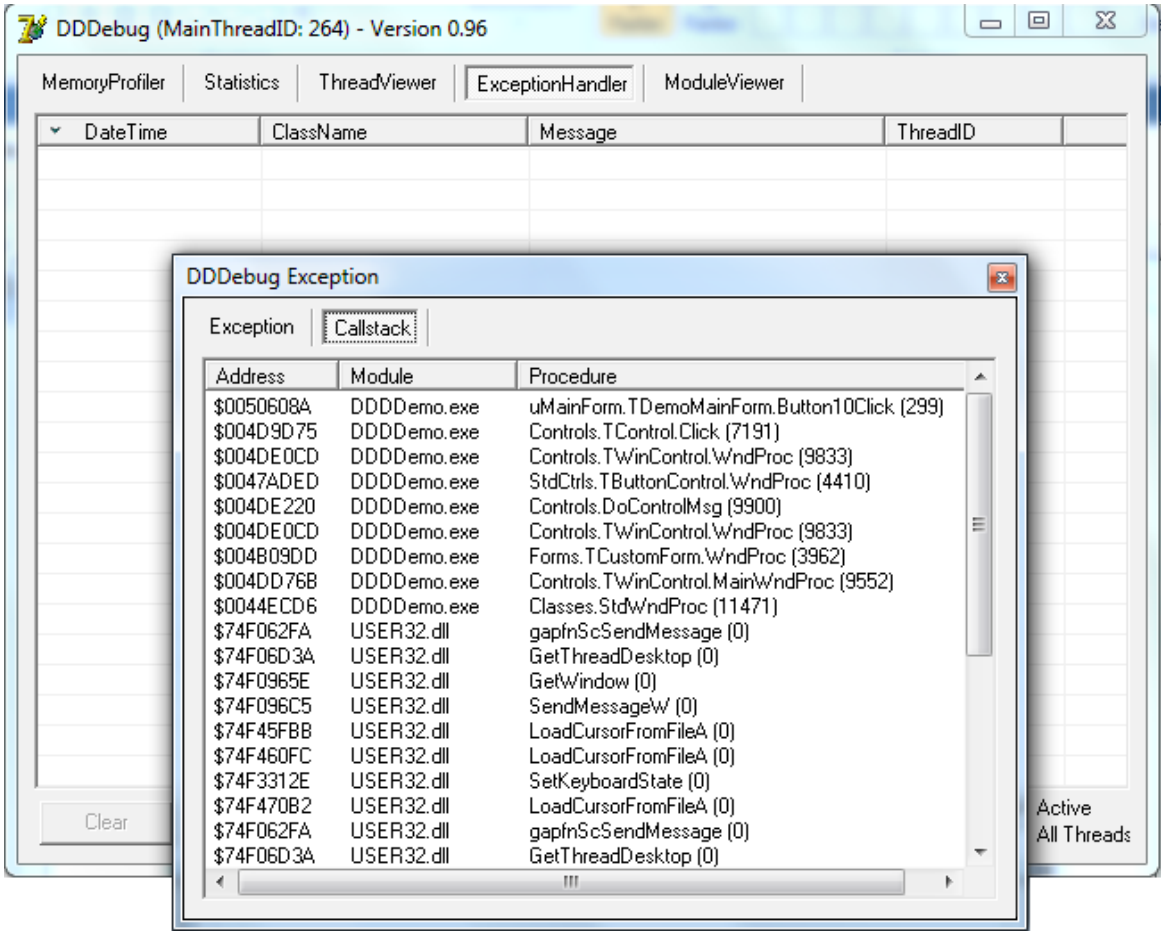
The Module Viewer

The view within the tab "ModuleViewer" displays a list of all loaded modules (DLLs and packages) of the process. Double-click on an entry to retrieve details of the selected module, a list of its exported functions as well as a list of modules the selected module depends on.



The Exception Handler

The view within the tab "ExceptionHandler" displays a list of raised exceptions which occurred during application run time. Double-click on an entry to view its details which included the class name of the exception, its source as well as the call stack of the exception as shown within the next screen shot.

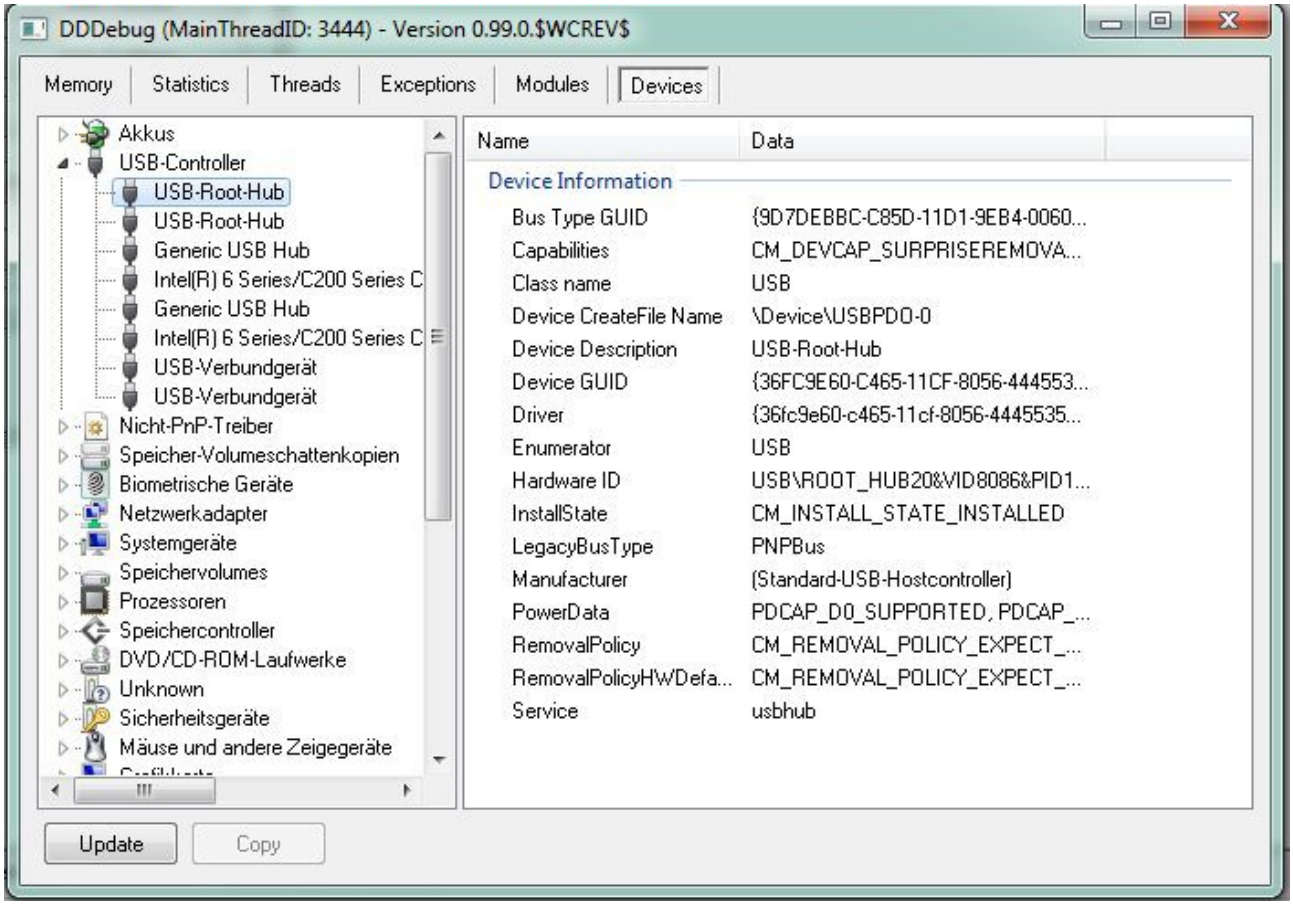


More features of DDDebug concerning exceptions are described within the section "Usage of the Exception Handler".

Please note the limitations of the trial version.

The Device Viewer

The view within the tab displays devices of the current machine in a tree like structure. Details of each device are displayed on the right pane as shown on the next screen shot.



Please note that the names of sections and devices are based on windows localization which is german in my case.

Advanced Topics

Finding prematurely freed Memory

Every programmer already has experienced access violations which have been caused by prematurely freed memory. These bugs sometimes are very hard to find as the error might have occured some long time before. DDDebug can help you isolate those problems with it's MemoryItemWatcher included within the unit DDDDebug.pas. The following snippet shows how to use the MemoryItemWatcher:

```
lWatcher := TDDDMemoryItemWatcher.GetInstance;
GetMemoryProfiler.Active := True;
lStrings := TStringList.Create;
try
 lWatcher.Watch(lStrings);
 // ...
 lWatcher.Unwatch(lStrings);
finally
 FreeAndNil(lStrings);
end;
```

If the watched string list will be freed somewhere within the call of Watch and Unwatch the MemoryItemWatcher will, depending on the value of its property UseBreakpoint, raise an exception or stop on a break point. In both cases, the debugger will stop to enable the programmer to isolate the problem.

The MemoryItemWatcher is also able to observe not only objects but also raw memory (GetMem) and interface references. Please note that the memory profiler needs to be active to use the MemoryItemWatcher.

This feature is not available within the trial version.

Finding Deadlocks

DDDebug is helpful finding deadlocks between threads at application run time utilizing the Wait Chain Traversal (WCT) API introduced in Windows Vista. Several classes encapsulate the Windows API in a Delphi like manner, making the usage straightforward.

The most common scenario, also included as test case within the DUnit tests does not need more than a couple of lines of code as well as the implementation of an event handle as shown within this code snippet:

```
LNotifier, TWaitChainNotifier.Create(DeadlockDetected);
LNotifier.CheckInterval := 500;
Lnotifier.Active := True

procedure TSomeClass.DeadlockDetected(Sender: TObject;
 AThreads: TDDDThreadInfoArray;
 var AAbortThreads, ATryTerminate: Boolean);
begin
 //
end;
```

The snippet above spawns a thread which periodically checks whether any deadlocks between the threads within the process do exist. The interval can be configured manually setting the property CheckInterval. As soon as any deadlock will be detected, an event - which can be set within the constructor or the property OnDeadlockDetected - will be invoked along with an array of TDDDThreadInfos (defined within DDDThreads.pas) of those threads involved within the deadlock. You can try to abort those threads (the abortion tries to use a cooperative way by raising an exception within the context of those threads and optionally, which is not recommended, uses TerminateThread), attach a break point here or dump the thread call stacks to a file.

The support of the WCT API will be enhanced within the next version of DDDebug to support more uncommon scenarios and to provide more information about the reason of the deadlock.

This feature requires Windows Vista and above.
This feature is not available within the trial version.

Usage of the Exception Handler

DDDebug contains an exception handler which traces all exceptions raised during application runtime, providing detailed information about the source of the exception, including the names of the units, classes and methods involved.

…

Additional Features of Delphi 2009 and above

Starting with Delphi 2009, the Delphi exception class contains two additional properties: StackTrace and StackInfo. These properties contain a readable stack trace but are not used or filled by default. The exception handler of DDDebug provides an easy way to use these properties and enlist them with these useful information:

```
ExceptionHandler.InitializeStackTraceProvider;
```

All exceptions raised hereafter will be automatically enriched with a human readable stack trace which includes the names of the units, classes and methods involved making it easy to migrate and enhance your application with this feature.

Please note the limitations of the trial version.

More about Filters

In addition to the default filter described within the section “Working with Filters”, some more filters are included and usable by the Memory Profiler. These filters need to be created and configured within your code.

These filters are:

TDDDTypeNameFilter	excludes classes and records by name
TDDDTypeNamesToTraceFilter	includes classes and records by name
TDDDThreadIdFilter	traces allocations of specified threads only

The following code snippet, which is based on the provided DUnit test cases, exemplifies the usage of these filters. The filter excludes disables the tracing of allocations of classes and records whose names start with TString. The syntax of the mask to be used can be looked up within the Delphi help (see TMask for more information).

```
lFilter := TDDDTypeNameFilter.Create;
lFilter.Add('TString*');
lFilter.Active := True;
```

Anonymous Methods

Anonymous methods, introduced in Delphi 2009, are implemented through interfaces which are generated by the compiler. The implementing objects are, usually, very short living and can be identified by containing the substring “$ActRec” in its class name.
As those objects are short living, the profiler trace resp. the memory profiler statistics, will only be poluted by those classes. Therefore, the memory profiler ignores them by default.
To trace those objects, you need to configure the default filter accordingly.

Remote Profiling

All of the above scenarios include the integration of the GUI of DDDebug. As it can not be integrated within applications which do not have a GUI, e.g. windows services, DDDebug provides another way to profile such applications: remote profiling. Remote profiling can also be used to observe memory allocations, retrieve thread callstacks etc. for applications running on another computer as well as applications executing on MacOS, iOS and Android as soon as DDDebug supports those operating systems.

Remote profiling is based on my remoting framework DDObjects.

The intregation, introduction and documentation of DDDebug Remote Profiling will be finished and added mid 2018.

DDObjects

DDObjects is a remoting framework to be used with Delphi 5 to 7 as well as Delphi 2005 to XE8 (32 and 64bit) which originally was begun out of personal interest for technologies like DCOM, RMI, Corba etc. A main goal in the development of DDObjects has not only been to keep the code one has to implement in order to utilize DDObjects as simple as possible but also very close to Delphi's usual style of event-driven programming.

DDObjects can be downloaded and obtained shortly at http://ddobjects.de/ddobjects

This feature is not available within the trial version and depends on DDObjects, which needs to be obtained separately.

Limitations of the Trial Version

Beneath the fact that the source code is not included there are limitations pertaining the number of allocations and exceptions to be processed by the profiler. As soon as 50,000 allocations have been traced, the memory profiler will turn inactive. As soon as 5 exceptions have been traced, the exception handler will turn inactive. The application needs to be restarted in order to trace allocations and exceptions again. Furthermore some features described above are not included within the trial version. Where applicable, this limitation is being mentioned.

Roadmap and future Enhancements

- Profiling of GDI and USER windows objects
- Remote and service application profiling
- Usability of GUI to be enhanced (icons, toolbars etc.)
- Enhancement and usability of memory profiler snapshots
- Customization of exception handling and reporting
- Support of Android, MacOS and iOS

Final Notes

Although DDDebug has been tested and used extensively on a wide variety of platforms, I am unable to test all supported Delphi versions in respect to their different service packs and hot fixes. If you are facing any problems, please let me know. Send an eMail to stefan@ddobjects.de supplying as much information (which version of Delphi and Windows are you using, which service packs and hot fixes are applied) as possible so I can try to reproduce and apply a fix.

Currently nothing else remains to say but

Enjoy! ;-)

Software License

As a proprietary product of Stefan Meisner, DDDebug is protected by copyright laws. At all times Stefan Meisner retains full title to the software. Subject to your acceptance of and accordance with the terms and conditions stated in this agreement, you shall be granted a single-user software license. Any previous agreement with Stefan Meisner is superseded by this agreement.

THIS SOFTWARE LICENSE GIVES YOU THE RIGHT TO

- Install and use the software for the sole purposes of designing, developing, testing, and deploying application programs which you create. You may install a copy of the software on two computers and freely move the software from one computer to another, provided that you are the only individual using the software.
- Write and compile your own application programs using the software.
- Make one copy of the software for backup or archival purposes or copy the software to a single permanent storage medium provided you keep the original solely for backup or archival purposes.
- Distribute the DDDebug runtime packages for the sole purpose of executing application programs created with Delphi.

ENGAGING IN ANY OF THE ACTIVITIES LISTED BELOW WILL TERMINATE THE SOFTWARE LICENSE. IN ADDITION TO SOFTWARE LICENSE TERMINATION, STEFAN MEISNER MAY PURSUE CRIMINAL, CIVIL, OR ANY OTHER AVAILABLE REMEDIES

- Distribution of any files contained in this software package, other than the runtime package explicitly listed above, including but not limited to .pas, .dfm, .dcu files, .dcp files, and design-time packages.
- Removal of proprietary notices, labels or marks from the software or documentation.
- Creation of an application that does not differ materially from the software.
- Creation of an application (whether it will be freeware, shareware or a commercial product) which competes directly or indirectly with DDDebug.
- Distribution of an application program created using the software to another developer. A developer is defined as any person who is executing an application program created using the software, on a computer which contains an installation of Delphi. In order to execute such an application, the developer must own a license to the software and must have installed the software on the computer.
- You may not use the software to create components to be used by other developers.

USE OF SOURCE CODE

Recipient will not utilize the source for the creation of software (whether it is freeware, shareware or a commercial product) which competes directly or indirectly with DDDebug. In addition, recipient will not disclose the source itself, nor the implementations discovered therein, to any party.

DISTRIBUTION OF SOURCE CODE

Recipient will not distribute the source. Specifically this includes all .dcu, .dfm, and .pas files which Stefan Meisner has provided.

TECHNICAL SUPPORT FOR SOURCE CODE

Stefan Meisner will not provide support for changes recipient makes to the source. Recipient assumes full responsibility for supporting any code or application which results from such modification. Recipient will not hold Stefan Meisner liable, directly or indirectly, for any changes made to the source, including changes which recipient has made based on advice or suggestions provided by Stefan Meisner.

SOURCE IS PROVIDED AS IS

Stefan Meisner makes no warranties, express or implied, with respect to the source and hereby expressly disclaims any and all implied warranties of merchantability and fitness for a particular purpose. In no event shall Stefan Meisner be liable for any direct, indirect, special, or consequential damages in connection with or arising out of the performance or use of any portion of the source.

LIMITED WARRANTY

Except as specifically stated in this agreement, the software and software documentation is provided and licensed "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

In no event will Stefan Meisner be liable to you for any damages, including without limitation lost profits or revenues, loss of data, business interruption loss, recovery or substitution costs, or claims by third parties, or other indirect incidental or consequential damages, arising out of the use or inability to use the software, even if Stefan Meisner has been advised of the possibility of such damages. In no case shall Stefan Meisner' liability exceed the amount of the license fee paid by you for the software.